

_confluent examples

20 декабря 2020 г. 0:46

<https://github.com/confluentinc/examples/tree/6.0.1-post/microservices-orders>

<https://docs.confluent.io/platform/current/tutorials/examples/microservices-orders/docs/index.html>

<https://github.com/confluentinc/qcon-microservices>

<https://drive.google.com/file/d/1vq1tcLk4FWK7BKRG5cEhOzbeXoy65hUK/view>

[Beyond Microservices: Streams, State and Scalability](#)

<https://github.com/confluentinc/kafka-streams-examples/tree/5.0.0-post/src/main/java/io/confluent/examplesstreams/microservices>

[Using Apache Kafka to implement event-driven microservices](#)

[Building Event Driven MicroServices with Apache Kafka](#)

[Building Event Driven Microservices with Stateful Streams](#)

[GOTO 2020 • Beyond Microservices: Streams, State and Scalability • Gwen Shapira](#)

[Beyond Microservices: Streams, State and Scalability](#)

[Papers We Love - QCon NYC Edition | Gwen Shapira on Realtime Data Processing at Facebook](#)

[Kafka and The Service Mesh | Gwen Shapira, Confluent](#)

[Scale by the Bay 2018: Gwen Shapira, Matthias Sax: Deploying Kafka Streams Applications...](#)

[Keynote: Gwen Shapira, Confluent | Kafka's New Architecture | Kafka Summit 2020](#)

[GOTO 2017 • Stream All Things - Patterns of Modern Data Integration • Gwen Shapira](#)

More Resources: Blogs and Docs

- <https://www.confluent.io/blog/building-a-microservices-ecosystem-with-kafka-streams-and-ksql/>
- <https://github.com/confluentinc/kafka-streams-examples/tree/5.0.0-post/src/main/java/io/confluent/examples/streams/microservices>
- <https://www.confluent.io/designing-event-driven-systems>
- <https://www.confluent.io/blog/build-services-backbone-events/>
- <http://docs.confluent.io/current/streams/concepts.html#duality-of-streams-and-tables>

More Resources: Video

- <https://www.infoq.com/presentations/microservices-arch-infrastructure-cd>
- Martin Fowler: <https://www.youtube.com/watch?v=STKCRSUsyP0>
- Trivadis: <https://www.youtube.com/watch?v=IR1NLfaq7PU>

If this excessively long article was not enough, and you want to read even more on the topic, I would recommend:

- My upcoming book, [Designing Data-Intensive Applications](#), systematically explores the architecture of data systems. If you enjoyed this article, you'll enjoy the book too.
- I previously wrote about similar ideas in 2012, from a different perspective, in a blog post called "[Rethinking caching in web apps](#)."
- Jay Kreps has written several highly relevant articles, in particular about [logs as a fundamental data abstraction](#), about the [lambda architecture](#), and about [stateful stream processing](#).
- The most common question people ask is: "but what about transactions?" — This is a somewhat

open research problem, but I think a promising way forward would be to layer a transaction protocol on top of the asynchronous log. [Tango](#) (from Microsoft Research) describes one way of doing that, and [Peter Bailis et al.](#)'s work on [highly available transactions](#) is also relevant.

- *Pat Helland has been preaching this gospel for ages. His latest CIDR paper [Immutability Changes Everything](#) is a good summary.*
- *There is a more applied guide to putting stream data to work in a company [here](#).*
- *Those in the Bay Area who are interested in learning more about Apache Samza should consider attending the [Samza meet-up](#) tonight.*

From <<https://www.confluent.io/blog/turning-the-database-inside-out-with-apache-samza/>>

для решения проблем отставания чтения своих же собственных данных в CQRS
Collapsing CQRS with a Blocking Read" on page 142 in Chapter 15, there are strategies for addressing this problem.

Kafka Streams API ships with a simple embedded database, called a state store, built into the API (see "Windows, Joins, Tables, and State Stores" on page 135 in Chapter 14).

We discuss this use of state stores for holding application-level state in the section "Windows, Joins, Tables, and State Stores" on page 135 in Chapter 14.

In Chapter 15 we walk through a set of richer code examples that create different types of views using tables and state stores, along with discussing how this approach can be scaled.

* ВОПРОСЫ

11 февраля 2021 г. 14:55

* макроархитектура

* макроархитектура (короткая версия)

BOOK Richardson Chris общий перечень всех аспектов

краткое объяснение

тезисы

- 1) **СВЕРИЛИ ТЕРМИНОЛОГИЮ** курс кафка / книга бена стопфорда / с полным примером и кодом
 - a. где достать курс по стримам
 - b. пример оказалось это главное
 - c. это книга как объяснение примера и как цикл статей для тех кто не хочет читать книгу
- 2) **ДА** много копий данных и не должны разъезжаться -> поэтому перезаливается
- 3) **НЕАВТОМАТИЧЕСКИ В ОБЫЧН БД** стейтфул процесминг подразумевает хранение своей версии обработанных данных, event carried state transfer данные наполняются в фоне
 - a. термин local state store (а не БД)
- 4) **НЕТ** данные шардируются и для клиента уже готовый набор собранный в фоне
- 5) **ДА** общение с внешкой по REST но внутри только кафка
 - a. только файлы+ЕСМ
 - b. еще локал БД может быть но вроде как должно быть rocksDB в kafka streams
 - c. а если БД есть то наполняется через kafka connect напрямую а не через микросервис
 - d. ..?еще
- 6) **ДА** балансировка через кафку consumer rebalancing (а не средствами кубера)
 - a. это как load balancing, service discovery кафка автоматически узнает о новом микросервисе

EVENT DRIVEN

20 декабря 2020 г. 11:22

* producer-consumer

23 февраля 2021 г. 13:11

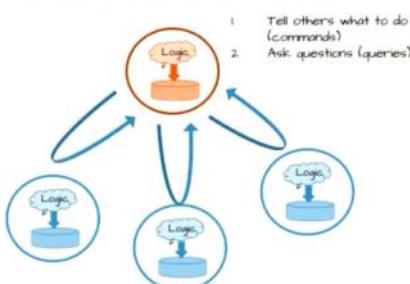
HTTP request-response pattern - цель-куда отправляем сообщение заранее известна

vs

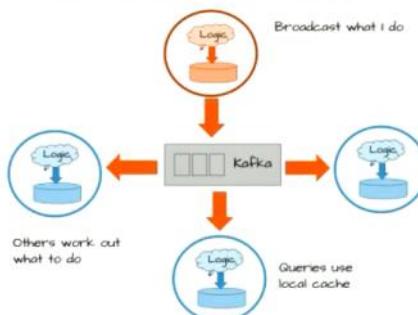
broadcast kafka pattern - каждое сообщение отправляем на всех, и все кто хочет читают

- таким образом я могу в кафке даже добавлять новые сервисы и это не как не зафектит уже существующие (а в REST паттерне не внося изменения в другие это вообще невозможно сделать)
- Обычно считается, что такие подходы обеспечивают лучшую масштабируемость и лучшую связку, чем их собратья типа запрос-ответ, поскольку они перемещают управление потоком от отправителя к получателю. Это увеличивает автономность каждой службы.
 - Это очень важное свойство архитектуры этого типа, называемое маршрутизацией, управляемой приемником Receiver Driven Routing. Логика передается получателю событий, а не отправителю. Бремя ответственности переворачивается! Передача управления приемникам снижает взаимосвязь между службами, что открывает важный уровень подключаемости к архитектуре: **теперь можно легко подключать новые микросервисы**. Компоненты можно легко менять местами.
- создание сервисов не с помощью цепочек команд, а, скорее, с помощью потоков событий. Это самостоятельный подход. Это также формирует основу для более продвинутых шаблонов, которые мы обсудим позже в этой серии,
- Обмен сообщениями помогает нам создавать слабосвязанные сервисы, потому что они перемещают чистые данные из сильно связанного места (источника) и помещают их в слабосвязанное место (подписчик). Таким образом, любые операции, которые необходимо выполнить с этими данными, выполняются не в источнике, а, скорее, на каждом подписчике, а технологии обмена сообщениями, такие как Kafka, снимают большую часть проблем с операционной стабильностью / производительностью.
- Передача сообщений решает эту проблему, разделяя отправителя и получателя. Реализация вычисления может распределять входящие сообщения по нескольким исполнительным блокам без необходимости информирования отправителя. Прелесть этого подхода в том, что семантика передачи сообщений не зависит от того, как получатель будет обрабатывать сообщение.

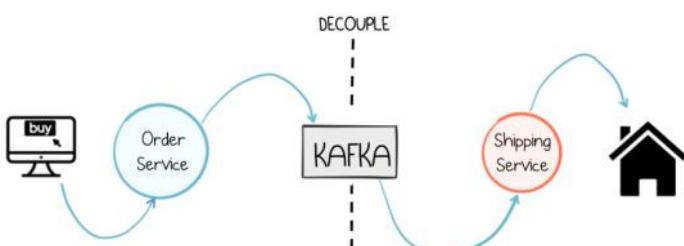
Request Driven



Event Driven

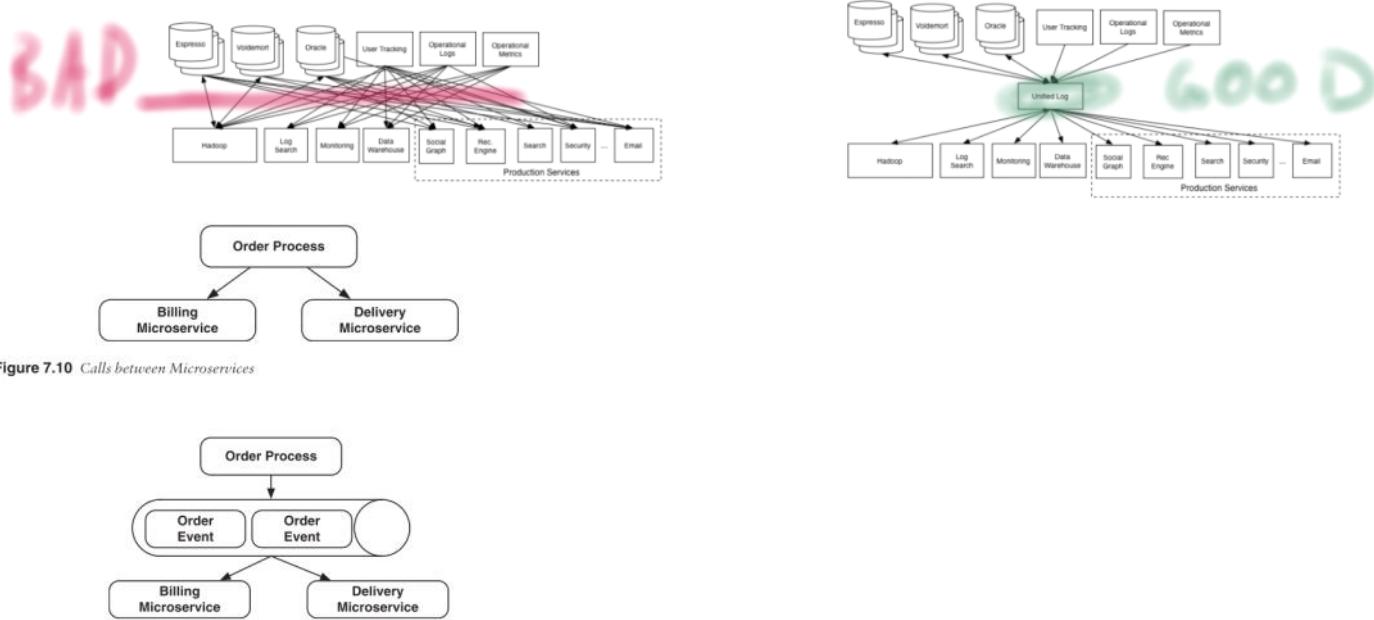


Synchronous request-response communication
Leads to
Tight point-to-point coupling



- Нам нужно было максимально изолировать каждого потребителя от источника данных. В идеале они должны интегрироваться только с одним хранилищем данных, которое дало бы им доступ ко всему.
- Идея состоит в том, что добавление новой системы данных - будь то источник данных или место

назначения данных - должно создать интеграционную работу только для подключения ее к единственному конвейеру вместо каждого потребителя данных.



в event-based подходе получатель события незвестен (а в message-based получатель известен)

- **Event-based systems** are typically organized around an **event loop**. Whenever something happens, the corresponding event is appended to a queue. The event loop continually pulls events from this queue and executes the callbacks that have been attached to them. Each callback is typically a small, anonymous procedure that responds to a specific event such as a mouse click. It may generate new events, which are then also appended to the queue and processed when their turn comes
- In contrast, **message-based systems** provide a way to send messages **to specific recipients**. The anonymous callback is replaced by an active recipient that consumes messages received from potentially anonymous producers. Whereas in an event-based system, event producers are addressable so that callbacks can be registered with them, in a message-based system the consumers are addressable so they can be given responsibility for processing certain messages. Neither the message producer nor the messaging system need concern itself with the correct response to a message; the current configuration of consumers determines that

преимущество данного подхода

- It allows processing to proceed sequentially for individual consumers, enabling stateful processing without the need for synchronization. This translates well at the machine level because consumers will aggregate incoming events and process them in one go, a strategy for which current hardware is highly optimized.
- Sequential processing enables the response to an event to depend on the current state of the consumer. So, previous events can have an influence on the consumer's behavior. A callback-based scheme, in contrast, requires the consumer to decide what its response will be when it subscribes to the event, not when the event occurs.
- Consumers can choose to drop events or short-circuit processing during system overload. More generally, explicit queueing allows the consumer to control the flow of messages. We will explain more about flow control in section 4.5.
- Last but not least, it matches how humans work in that we also process requests from our coworkers sequentially.
 - этот пункт может вас удивить, но мы находим знакомые ментальные образы полезными для визуализации поведения компонента
- обработка событий может быть преостановлена (если потребитель недоступен) и легко продолжена

Когда получатель анонимен, гарантировать доставку с подтверждением сложно

- В этом случае неясно, кто должен получить сообщение и есть ли вообще какие-либо получатели, которые должны получить сообщение. Поэтому также неясно, кто должен выдавать квитанции Guaranteed Delivery.

Доставка, а также обработка сообщения почти всегда могут быть гарантированы так как сообщения хранятся долго.

- Обработка обеспечивается, например, тем, что получатели подтверждают сообщение.
- Таким образом, асинхронная связь решает проблемы, вызванные распределенными системами.
- Kafka предлагает возможность хранить сообщения в течение очень долгого времени. Это может быть полезно для event sourcing

producer может класить сообщения независимо ни от кого, а consumer может читать сообщения независимо ни от кого

- [onenote: REST.one#Tell,%20Don't%20Ask%20принцип%20асинхронности§ion-id={A351751F-B444-4057-93FB-D671AB1B0B21}&page-id={DA5B8473-BAF0-4F8D-817C-02FCC0BA9822}&end&base-path=C:\Users\trans\Qsync\vova_from_onenote\lf_aglone_v1\SYSTEMDESIGN\KAFKA](#)
 - при синхронном общении обе стороны должны быть готовы к общению одновременно.
 - При асинхронной связи отправитель может отправлять независимо от того, готов получатель или нет
- Другими словами, асинхронная передача сообщений означает, что получатель в конечном итоге узнает о новом входящем сообщении и затем потребляет его, как только потребуется. Есть два способа информирования получателя:
 - он может зарегистрировать обратный вызов, описывающий, что должно произойти в случае

- определенного события,
- или он может получить сообщение в почтовом ящике (также называемом очередью)
- Создание асинхронной и неблокирующей связи вместо вызова синхронных методов позволяет получателю выполнять свою работу в другом контексте выполнения, например в другом потоке**

событие имеет значение вне контекста конкретного взаимодействия между его производителем и потребителями.

- Событие имеет значение, которое не зависит от его производителя и его потребителей, и в результате производители событий и потребители событий могут быть полностью отделены друг от друга. An event has meaning that is independent of its producer and of its consumers, and as a result event producers and event consumers can be completely decoupled from each other
- Идея использования самого события для разделения производителя событий и потребителя событий является существенным различием между программированием, основанным на событиях, и дизайном приложения, основанным на взаимодействиях запроса и ответа. The idea of using the event itself to decouple the event producer and event consumer is a significant difference between event-based programming and application design based on request-response interactions.
- В развязанной системе может быть более одного потребителя события, и предпринимаемые действия, если вообще предпринимаются какие-либо действия, могут значительно различаться между потребителями. Он также может меняться в течение всего срока службы приложения. Поскольку производитель событий не знает, что потребитель событий собирается делать с событием, и даже не знает, сколько их потребителей, для производителя событий обычно нет смысла ожидать ответа на свои события.

однако коньюмер зависит от продьюсера (несмотря на то что продьюсер не зависит от коньюмера)

- Когда вы реализуете поставщика услуг, вы обычно кодируете его так, чтобы он предоставлял эту конкретную услугу, независимо от характера или цели запрашивающей услуги, поэтому поставщик отделен от запрашивающей стороны. Однако запрашивающая услуга зависит от поставщика услуг, выполняющего согласованную функцию.

Потребитель не отправляет ответ производителю

- кроме, возможно, указания на то, что он получил событие. Это означает, что потребитель может обрабатывать событие асинхронно производителю.

receive event from producer

Большинство брокеров сообщений предоставляют возможность публикации-подписки, в которой логика маршрутизации сообщений определяется получателями, а не отправителями; этот процесс известен как маршрутизация, управляемая получателем. Таким образом, получатель сохраняет контроль над своим присутствием при взаимодействии, что делает систему подключаемой

нужно добавить новый микросервис - легко добавить новый микросервис (изменения нужно будет делать только в одном месте)

- Преимущество хореографических систем в том, что они подключаемые. Если платежная служба решает создать три новых типа событий для платежной части рабочего процесса, пока остается событие «Платеж обработан», она может сделать это, не затрагивая другие службы. Это полезно, потому что это означает, что если вы реализуете службу, вы можете изменить способ своей работы, и никакие другие службы не должны знать об этом или заботиться об этом. Напротив, в оркестрованной системе, где рабочий процесс диктуется одной службой, все изменения необходимо вносить в контроллер. Какой из этих подходов лучше всего зависит от варианта использования, но преимущество оркестрации в том, что весь рабочий процесс записывается в коде в одном месте.

различие между запросом и событием, как мне кажется, кроется в понятии сессии

- Запросы представляют собой части более крупного процесса взаимодействия (сессии).
- События видятся мне более «одноразовыми», асинхронными по своей природе. События важны и должны соответствующим образом обрабатываться, но они оказываются вырваны из основного контекста взаимодействия и ответ на них приходит лишь спустя некоторое время
- А так как каждое событие концептуально независимо от остальных, вынужденное ослабление связей внутри системы, построенной на основе функций, позволяет снизить ее концептуальную сложность, позволяя разработчику сосредоточиться на шагах, необходимых для обработки только одного конкретного типа событий.

Event Sourcing отличается от команд из CQRS.

- Команды специфичны: они точно определяют, что нужно изменить в целевом объекте.
- События содержат информацию о том, что случилось

If an event is a record of an occurrence **in the past**, then a command is the expression of intent that a new event will occur **in the future**

- Команда является приказом или указанием на конкретное действие, которое будет выполняться в будущем
- почти все программное обеспечение полагается на неявные команды - решение сделать что-либо немедленно сопровождается выполнением этого решения в одном блоке кода.
- Если событие представляет собой запись о произшествии в прошлом, тогда команда является выражением намерения, что новое событие произойдет в будущем
- Грамматически: событие - это глагол в прошедшем времени в изъявительном наклонении, а команда - это глагол в повелительном наклонении .

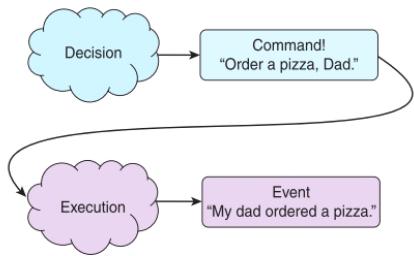


Figure 9.1 A decision produces a command—an order or instruction to do something specific. If that command is then executed, we can record an event as having occurred.

свойства команды

- **Shrink-wrapped**—The command must contain everything that could possibly be required to execute the command; the executor should not need to look up additional information to execute the command. команда должна содержать все, что может потребоваться для выполнения команды; исполнителю не нужно искать дополнительную информацию для выполнения команды.
 - Исполнитель команд должен иметь право выбирать, как лучше всего выполнять каждую команду, но исполнителю не нужно искать дополнительные данные или содержать бизнес-логику, специфичную для команды
- **Fully baked**—The command should define exactly the action for the executor to perform; we should not have to add business logic into the executor to turn each command into something actionable. команда должна точно определять действие, которое должен выполнить исполнитель; нам не нужно добавлять бизнес-логику в исполнитель, чтобы превратить каждую команду во что-то действенное

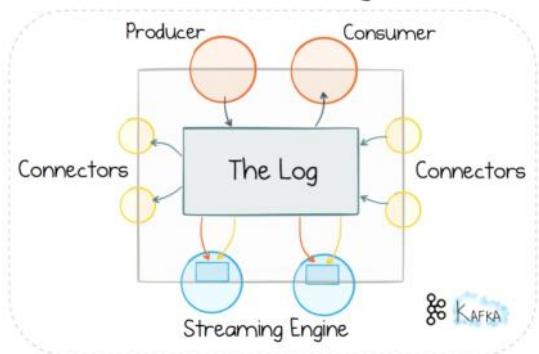
* centralize+decentralize

23 февраля 2021 г. 13:07

Decentralized Approach используя kafka: streaming platform

- broker is a distributed log, rather than a traditional messaging system, Мы можем использовать для этого распределенный журнал. В основе Apache Kafka лежит распределенный журнал
- те распределенный лог - это просто механизм которым все это обеспечивается
- компоненты стриминговой платформы
 - Java API
 - коннекторы/адаптеры
 - трансформаторы KSQL/Kstreams Java API(+stateful local store API)

Kafka: a Streaming Platform

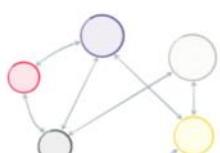


centralized stream of events

decentralized stateful stream processing

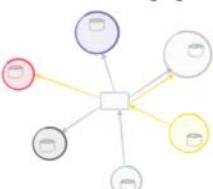
- центральный слой кафки и query layer поверх
- вокруг кафки куча непрерывно обновляемых materialized view
- те конкретная выюха сделана только под конкретную задачу, что полностью соответствует концепции микросервисов (где каждый микросервис тоже сделан под конкретную задачу)
- те конкретная выюха сделана только под конкретную задачу (но у микросервиса выюхи может и не быть, если задача не стоит), что полностью соответствует концепции микросервисов (где каждый микросервис тоже сделан под конкретную задачу)
- Centralize an immutable stream of facts. Decentralize the freedom to act, adapt, and change.

(A) Service Interfaces



Data & Function in Owning Services

(B) Messaging



Data & Function Everywhere

(C) Shared Database

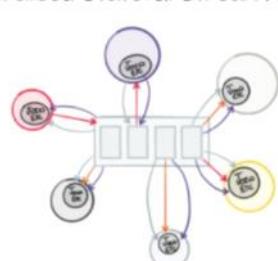


Data & Function Centralized

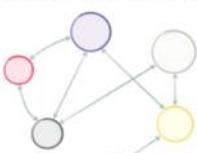
Option (D)

Centralized Stream of Events

Decentralised Stateful Stream Processing



(A) Service Interfaces



Data & Function in Owning Services

(B) Messaging



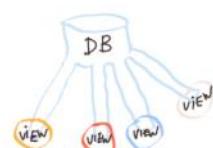
Data & Function Everywhere

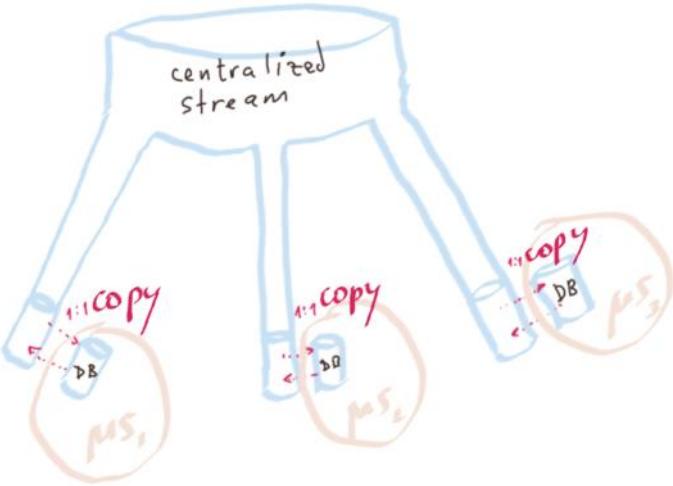
(C) Shared Database



Data & Function Centralized

(D) streaming platform





central data-plane которая содержит общие наборы данных и поддерживает синхронизацию сервисов, (чтобы данные не разъезжались между собой)

- цель кафки синхронизировать все данные во всех микросервисах
- Такие стриминговые сервисы не сомневаются, они не останавливаются. Они сосредотачиваются на «сейчас» - изменениях, перенаправлении и реформировании; ветвление субпотоков, преобразование таблиц, изменение ключей для перераспределения и повторное объединение потоков. Таким образом, это модель, которая охватывает параллелизм не через грубую силу, а вместо этого, ощущая естественный поток системы и трансформируя его по своему усмотрению.

Centralize an immutable stream of facts

Decentralize the freedom to act, adapt, and change.

- централизовать неизменный поток фактов. Децентрализовать свободу действий, адаптации и изменений

**Centralize an immutable stream of facts.
Decentralize the freedom to act, adapt and change.**

кафка уже делает большую часть того что обычно надо делать для построения распределенного приложения (поэтому явным образом использовать Zookeeper и программировать алгоритм консенсуса мне уже ненадо)

- Обеспечение согласованности данных (будь то конечное или немедленное) путем упорядочения одновременных обновлений узлов
- Обеспечить репликацию данных между узлами
- Предоставить писателю семантику "фиксации" (т.е. подтверждение только тогда, когда ваша запись гарантированно не будет потеряна)
- Предоставьте канал подписки на внешние данные из системы
- Предоставлять возможность восстанавливать отказавшие реплики, которые потеряли свои данные, или запускать новые реплики.
- Обработка перебалансировки данных между узлами.

На самом деле это существенная часть того, что делает распределенная система данных. Фактически, большая часть того, что осталось, относится к окончательному API запросов, ориентированному на клиента, и стратегии индексации. Это именно та часть, которая должна варьироваться от системы к системе: например, для запроса полнотекстового поиска может потребоваться запросить все разделы, тогда как

запрос по первичному ключу может потребовать запроса только одного узла, ответственного за данные этого ключа.

кафка предоставляет уже готовый слой абстракции, поэтому создавать системы быстрее (и бизнес получит отдачу быстрее)

- Использование программного обеспечения для обработки событий может существенно снизить общую стоимость владения приложением для обработки событий. Снижение затрат происходит за счет уровня абстракции и аналогично преимуществу, которое вы получаете от использования системы управления базами данных для хранения данных, а не от файловой системы. Программное обеспечение для обработки событий обычно предоставляет абстракции для обработки событий, которые находятся на более высоком уровне, чем те, которые предоставляются традиционными языками программирования. Это может снизить стоимость разработки и обслуживания и, следовательно, общую стоимость владения. В некоторых случаях эти абстракции более высокого уровня могут также позволить политехническим специалистам создавать правила обработки событий. Мы вернемся к этому моменту в главе 12, обсуждая обработку событий завтрашнего дня.

Еще одна проблема эффективности - это гибкость бизнеса. Вам может потребоваться относительно быстро изменить функциональность обработки событий, и гораздо проще вносить быстрые изменения в функции, которые выражаются с помощью абстракций программирования более высокого уровня и которые отделены от основной логики приложения

* метафора потока/стрима

23 февраля 2021 г. 13:25



ALL YOUR DATA
IS
EVENT STREAMS

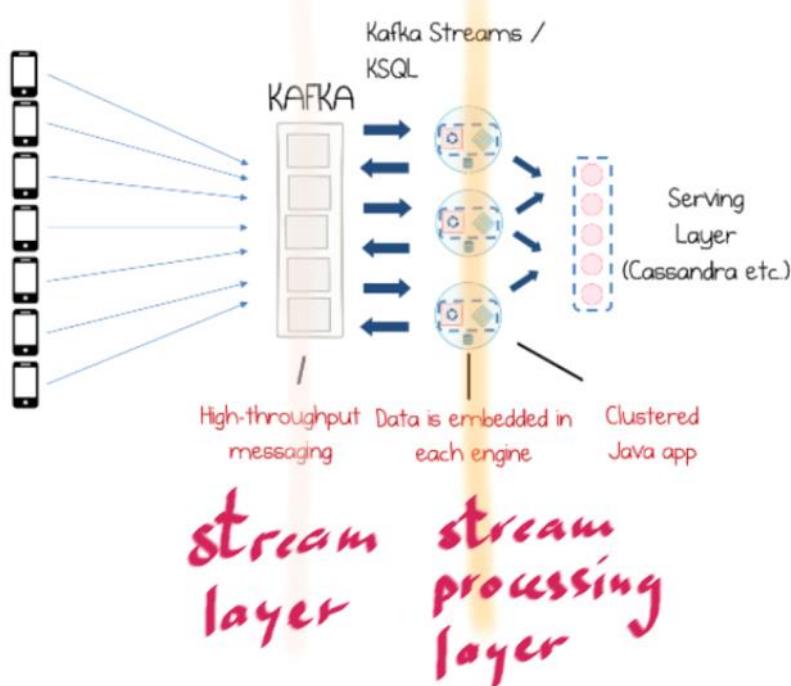
метафора потока/стрима

stream processing - обработка фактов "по одному но постоянно" в реальном времени real time data, по мере их поступления нам нужно думать не о запросах а об объединении потоков данных (хотя KSQL похож на SQL)

- continuously method of data collection - те как данные приходят с самого начала от источника данных то в таком ритме

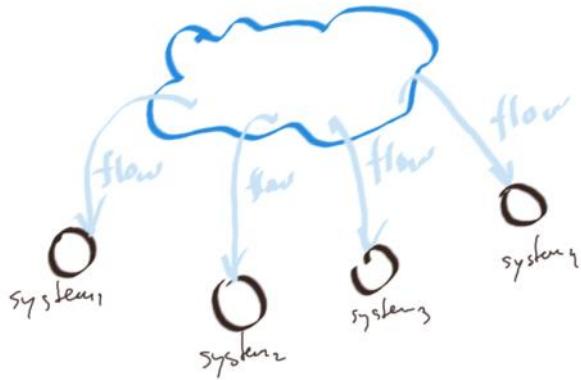
непрерывно (а НЕ периодически) они и обрабатываются

- Более плодотворный подход состоит в том, чтобы брать потоки фактов по мере их поступления и функционально обрабатывать их в реальном времени.
- Что касается чтения, нам нужно меньше думать о запросах к базам данных и больше о потреблении и объединении потоков, а также поддержании материализованных представлений данных в той форме, в которой мы хотим их читать.
- Пространство больших данных столкнулось с аналогичными проблемами, поскольку наборы данных размером в несколько терабайт подчеркнули неотъемлемую непрактичность пакетного управления данными., что, в свою очередь, привело к повороту в сторону стриминга.
- В потоковой модели нет общей базы данных. База данных - это поток событий, и приложение просто преобразует его во что-то новое.
 - Честно говоря, потоковые системы по-прежнему имеют атрибуты, похожие на базы данных, такие как таблицы (для поиска) и транзакции (для атомарности), но этот подход имеет радикально иное ощущение, больше похожее на функциональные языки или языки потока данных (и существует много перекрестного опыта между сообщества потокового и функционального программирования).
- В отличие от традиционной базы данных, эти **запросы выполняются непрерывно**, поэтому каждый раз, когда входные данные поступают на уровень обработки потока, запрос повторно вычисляется, и выдается результат, если значение запроса изменилось. После того, как новое сообщение прошло через все потоковые вычисления, результат попадает на обслуживающий уровень, из которого его можно запросить.
- Я рассматриваю потоковую обработку как нечто гораздо более широкое: инфраструктуру для непрерывной обработки данных - **continuous data processing**
- Настоящей движущей силой модели обработки является **метод сбора данных**. Данные, которые собираются в пакетном режиме **in batch (периодически)**, естественно, обрабатываются пакетно. Когда данные собираются непрерывно **continuously**, они, естественно, обрабатываются непрерывно
 - Перепись в США является хорошим примером пакетного сбора данных. **Периодически** начинается перепись, которая обнаруживает и переписывает граждан США методом перебора, заставляя людей ходить от двери к двери. Это имело большой смысл в 1790 году, когда впервые началась перепись. Сбор данных в то время был изначально ориентирован на партии, он включал в себя езду верхом и запись записей на бумаге, а затем транспортировку этой партии записей в центральное место, где люди складывали все подсчеты. В наши дни, когда вы описываете процесс переписи, сразу возникает вопрос, почему мы не ведем журнал рождений и смертей и не производим подсчет населения постоянно или с любой необходимой степенью детализации.



stream of events = log structured data flow (так как бы поток данных текущий в разные системы, **Data integration**)

- Журнал - это естественная структура данных для управления потоком данных между системами. Рецепт очень простой: Возьмите все данные организации и поместите их в центральный журнал для подписки в реальном времени
- см также выше "(a) dataflow style и functional style лучше всего подходят для distributed design"
- По моему мнению, **цель интеграции данных — гарантировать, что данные попадают во все нужные места**, будучи представленными в нужной форме. Для этого следует прочитать входные данные, преобразовать их, объединить, отфильтровать, агрегировать, применить модели обучения, оценки и, наконец, записать результаты. Инструментами для достижения цели являются пакетные и потоковые процессы. Результатами пакетных и потоковых процессов являются производные наборы данных, такие как поисковые индексы, материализованные представления, рекомендации для пользователей, совокупные показатели и т. п.
- [onenote:KAFKA%20STREAMS.one#stream%20\%20table%20\%20topology§ion-id={422214CB-B11E-4F4F-93A0-EF7DDD1C10A7}&page-id={AEBFB62D-8F1A-4548-86E9-FAF6458FAF09}&end&base-path=C:\Users\trans\Qsync\vova_from_onenote\tf_agonote_v1\SYSTEMDESIGN\KAFKA](#)



- Collects events from disparate source systems
- Stores them in a *unified log*
- Enables data processing applications to operate on these event streams

принцип dual writes (как ETL)

- Поскольку одни и те же или связанные данные присутствуют в нескольких местах, их необходимо синхронизировать: если какой-либо элемент обновляется в БД, то должен обновиться также в кэше, индексах поиска и складе данных
- С помощью последних эта синхронизация данных обычно выполняется благодаря процессам ETL
- двойная запись, при которой **в случае изменения данных приложение явно записывает информацию в каждую из систем**; например, сначала заносит данные в базу, затем обновляет поисковый индекс, а потом аннулирует записи кэша (или даже выполняет все эти операции одновременно).

известные проблемы dual writes

- состояние гонки. Из-за неудачного хронометража запросы чередуются: база сначала принимает запись от клиента 1 и устанавливает значение А, а затем — запись от клиента 2 и устанавливает значение В, вследствие чего окончательное значение элемента в базе равно В. Поисковый индекс, наоборот, сначала видит запись от клиента 2, а затем — от клиента 1, так что конечным значением в индексе поиска является А. Теперь эти две системы несовместимы, даже несмотря на отсутствие ошибки
- Другая проблема с двойной записью заключается в том, что одна из записей может оказаться неудачной, а вторая — успешной. Это уже проблема не конкурентного доступа, а отказоустойчивости, но она также приводит к несовместимости двух систем. Гарантировать то, что обе записи были либо успешными, либо нет, — задача атомарной фиксации (**или SAGA**), и ее решение стоит дорого

derived data systems - потребителей журнала можно назвать производными информационными системами:

- [source of truth -> derived data systems](#)
 - данные, хранящиеся в поисковом индексе и на складе, — просто другое представление данных из системы записи
- что такое event processing (из одного потока рождаем другой)

- В простейшем случае обработка событий включает чтение одного или нескольких событий из потока событий и выполнение каких-либо действий с этими событиями. Эта операция обработки может заключаться в фильтрации события из потока, проверке события по схеме или обогащении события дополнительной информацией. Или мы могли бы обрабатывать несколько событий одновременно, возможно, с целью их переупорядочения или создания какой-то сводки или агрегата этих событий.
- Приложение будет обрабатывать необработанный поток событий Nile как свой собственный входной поток, а затем генерирует выходной поток событий на основе этих входящих событий.
- Мы можем сказать, что любая программа или алгоритм, который понимает упорядоченную по времени природу непрерывного потока событий только с добавлением и может потреблять события из этого потока значимым образом, может обрабатывать этот поток.

Общий термин для обозначения того, что делают все эти примеры приложений, - это обработка потока событий

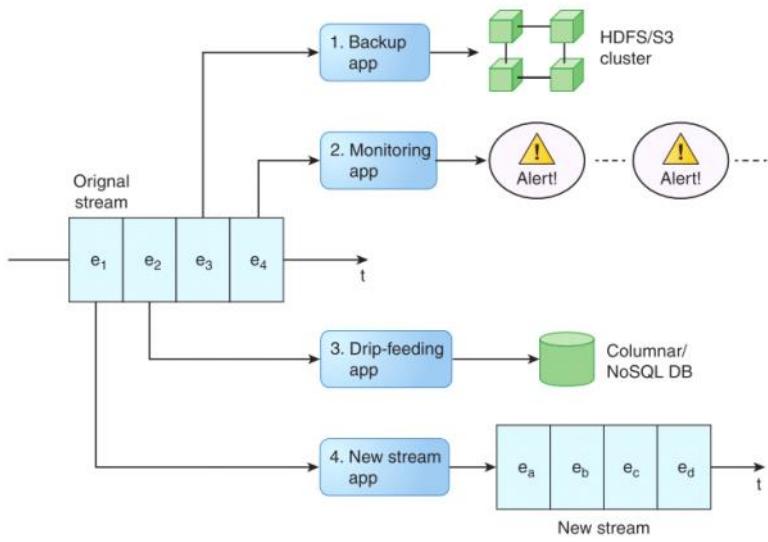


Figure 3.1 Our original event stream is being processed by four applications, one of which is generating a new event stream of its own. Note that each of our four applications can have a different current offset, or *cursor position*, in the original stream; this is a feature of unified logs.

"log" is another word for "stream" and logs are at the heart of stream processing.

в целом непрерывный процесс обработки данных более равномерно распределяет нагрузку и в итоге получается быстрее (чем периодический batch processing)

- Но по мере того как эти процессы заменяются непрерывными потоками, естественным образом начинается переход к непрерывной обработке, чтобы сгладить необходимые ресурсы обработки и уменьшить задержку

те для того чтобы все это заработало нужно собирать данные в реальном времени (как в реактивном программировании)

- Я думаю, что отсутствие сбора данных в реальном времени, вероятно, обрекло коммерческие системы потоковой обработки. Их клиенты по-прежнему занимались файловой, ежедневной пакетной обработкой для ETL и интеграции данных. Но оказалось, что в то время очень немногие люди действительно имели потоки данных в реальном времени
- Когда данные собираются партиями, это почти всегда происходит из-за некоторого ручного действия или отсутствия оцифровки или является историческим реликтом, оставшимся от автоматизации какого-либо нецифрового процесса
- Рабочие «пакетные» задания на обработку, которые выполняются ежедневно, часто эффективно имитируют вид непрерывных вычислений с размером окна в один день.

акцент паттерна event stream на realtime транспортировке событий

- Потоки событий также могут транспортировать информацию между различными компонентами
- Шаблон потока событий использует эти постоянные события изменений для реализации надежного и масштабируемого распространения информации по всей системе, не обременяя исходные объекты домена этой задачей. Созданные события могут быть распределены поддерживающей инфраструктурой и использоваться любым количеством заинтересованных клиентов для получения новой информации из их комбинации или для поддержания денормализованных представлений данных, оптимизированных для запросов.

полный realtime в event stream не подразумевается так как сначала сообщение нужно заперсистить в логе что уже занимает некоторое время (и кроме того event это обычно уже случившийся факт те уже в прошлом)

- Важным свойством потоков событий является то, что они не представляют текущее состояние объекта системы: они состоят только из неизменяемых фактов о прошлом, уже были сохранены в постоянном хранилище. Компоненты, которые генерируют эти события, могут перейти в более новые состояния, которые позже будут отражены в потоке событий. Задержка между отправкой события и распространением потока зависит от качества реализации журнала, но тот факт, что существует значительная задержка, является неотъемлемой частью этой архитектуры, и ее нельзя избежать
- Во всех случаях, когда временная задержка и ограничения согласованности не являются проблемой, предпочтительно полагаться на этот шаблон вместо того, чтобы тесно связывать источник изменений с его потребителями. Шаблон потока событий обеспечивает надежное распространение информации по всей системе, позволяя всем потребителям выбирать желаемую надежность, поддерживая смещения чтения и сохраняя свое состояние там, где это необходимо. Самым большим преимуществом является то, что это надежно помещает

источник истины в одном месте - в журнале - и устраниет сомнения относительно того, из какого места могут быть получены различные фрагменты информации

Люди склонны больше доверять системам, в которых поток является явным.

- Даже если поток не нужно определять явно, он должен быть видимым и обновляемым. Это мнение основано на опыте и отзывах пользователей.

Designing for auditability - аудитопригодная архитектура

- Благодаря явному представлению информационного потока происхождение данных становится более понятным; это делает более выполнимой проверку целостности.
- Чтобы убедиться в отсутствии повреждений хранилища событий, можно использовать кэши журнала событий.
- Для любого производного состояния можно повторно запустить пакетные и потоковые процессоры, которые создали его на основе журнала событий, и проверить, получим ли мы тот же результат, или даже запустить параллельно создание избыточного производного состояния.
- Детерминированный и четко определенный информационный поток также облегчает отладку и отслеживание выполнения системы с целью определить, почему она работает так, а не иначе [4, 69].
- При неожиданном развитии событий важно иметь доступ к проведению диагностики, уметь воспроизводить точные обстоятельства, которые привели к этому событию, — своего рода возможность отладки во времени.
-

state machine replication

- [state machine replication](#) -
- если каждое сообщение представляет собой запись в базу и каждая реплика обрабатывает одни и те же записи в той же последовательности, то реплики будут оставаться согласованными между собой
- В подразделе «Рассылка общей последовательности» раздела 9.3 мы также столкнулись с принципом репликации конечных автоматов, который гласит: если каждое событие представляет собой запись в базу данных и каждая реплика обрабатывает одни и те же события в одном и том же порядке, то все реплики будут завершены в том же конечном состоянии. (Обработка события считается неделимой операцией.) Это просто еще один случай потоков событий

shared nothing микросервисы (потому что они связаны только евентами)

This idea was extended by microservice implementers, so a bounded context describes a set of closely related components or services that share code and are deployed together. Across bounded contexts there is less sharing (be it code, functionality, or data). In fact, as we noted earlier in this chapter, microservices are often termed “shared nothing” for this reason.⁶

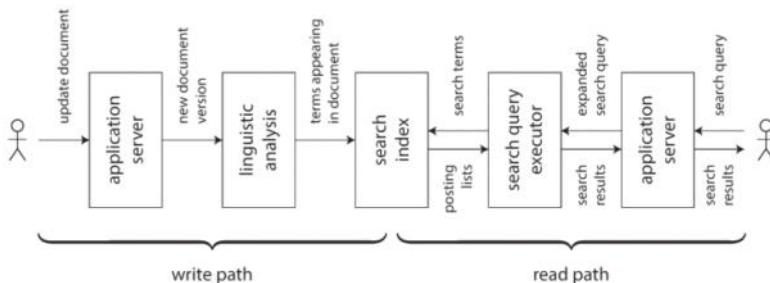
Sharing nothing simplifies concurrency

- порядок, в котором сообщения от Алисы и Боба достигают Чарли, не определен, если Алиса и Боб не приложат значительных усилий для синхронизации своего взаимодействия⁵. Ответ на вопросы Боба. - ion: «Вы уже слышали от Алисы?» таким образом, будет непредсказуемо различаться при разных исполнениях этого сценария.
- Точно так же, как распределение влечет за собой параллелизм, верно и обратное. Параллелизм означает, что два потока выполнения Могут выполняться одновременно, независимо друг от друга. В неидеальном мире это означает, что оба потока могут выходить из строя независимо, что делает их распределенными по определению.
- Следовательно, всякий раз, когда система содержит параллельные или распределенные компоненты, взаимодействие между этими компонентами будет недетерминированным.
- В этом смысле термин параллелизм без совместного использования означает, что внутреннее изменяемое состояние каждого модуля

безопасно хранится внутри него и не используется напрямую с другими модулями. Примером того, что запрещено, является отправка ссылки на изменяемый объект (например, массив Java) из одного модуля в другой при сохранении ссылки. Если оба модуля впоследствии изменят объект в пределах своей транзакции, их логика будет сбита с толку из-за дополнительной связи, которая не имеет ничего общего с передачей сообщений.

- Повторяя наши шаги, мы видим, что переход от параллелизма с общим состоянием к параллелизму без общего доступа устраниет цели Й класс проблем из области приложения. Нам больше не нужно беспокоиться о том, как процессоры синхронизируют свои действия, потому что мы пишем инкапсулированные компоненты, которые отправляют только неизменяемые сообщения, которые можно совместно использовать без проблем. Освободившись от этого беспокойства, мы можем сосредоточиться на сути распределенного программирования для решения наших бизнес-задач, и именно здесь мы хотим быть в случае необходимости распространения.
- Ключевой момент, развитый в предыдущих главах, заключается в том, что модули взаимодействуют только асинхронно, передавая сообщения. Они не разделяют изменяемое состояние напрямую.

write path + read path



Вместе путь записи и путь чтения формируют весь путь данных от пункта их сбора до пункта потребления (возможно, другим человеком)

write path

- Путь записи — та часть пути, где данные вычисляются предварительно; другими словами, это делается максимально быстро в момент поступления данных, независимо от того, поступил ли запрос на них

read path

- Путь чтения — часть маршрута, которая выполняется только тогда, когда кто-то сделал запрос. Если вы знакомы с функциональными языками программирования, то могли заметить, что путь записи похож на упреждающие вычисления, а путь чтения — на отложенные

derived dataset

- Производный набор данных — точка соединения пути записи и пути чтения, как показано на рис. 12.1. Он представляет собой компромисс между объемами работы, которые должны быть выполнены во время записи и чтения соответственно

пример

- Если бы индекса не было, то поисковый запрос должен был бы сканировать все документы (как команда grep), что в случае большого количества документов потребовало бы значительных вычислительных затрат. Отсутствие индекса означает меньший объем работы на пути записи (индекс не обновляется), но гораздо больший — на пути чтения
- И наоборот, можно представить предварительные вычисления результатов поиска для всех вероятных запросов. В этом случае меньше работы на пути чтения: никакой булевой логики, просто найти результаты для своего запроса и выдать их пользователю. Однако путь записи будет намного дороже: набор всех возможных поисковых запросов бесконечен и, таким образом, для предварительного вычисления всех потенциальных результатов поиска потребуется неограниченное время и пространство для их хранения, так что это не будет работать

Я считаю, что идея границы между путями записи и чтения заслуживает внимания, поскольку можно обсуждать смещение этой границы и исследовать значение данного сдвига на практике

- граница между путями записи и чтения может быть разной для популярных и обычных пользователей

? разные акценты в корпоративных и интернет app

<u>Enterprise SW:</u>	<u>Internet companies:</u>
biggest problem: complexity of domain ⇒ business logic	biggest problem: volume of data ⇒ complexity of data infrastructure

Event sourcing	Stream processing
---------------------------	------------------------------

* обработка событий

23 февраля 2021 г. 13:28

The seven fundamental building blocks of event processing

- Агент обработки событий(это брокер) строительный блок представляет собой часть промежуточной логики обработки событий , вставленной между производителями и потребителями событий событий.
- канал событий в основной задачей является событий маршрута между производителями и потребителями событий
- Контекст элемент собирает набор условий различных размеров (временная, пространственная, сегментация-ориентированный, и состояние-ориентированное), что дает нам возможность категоризации событий экземплярам , так что они могут быть направлены на соответствующие экземпляры агентов. Например, вы можете использовать контекст, ориентированный на сегментацию, чтобы гарантировать, что события, относящиеся к разным клиентам, обрабатываются разными экземплярами агента обработки событий.
- A global state (это справочники) element refers to data that is available for use both by event processing agents and by contexts. This data may be system-wide global variables, reference data used to enrich events, and event stores that hold past events

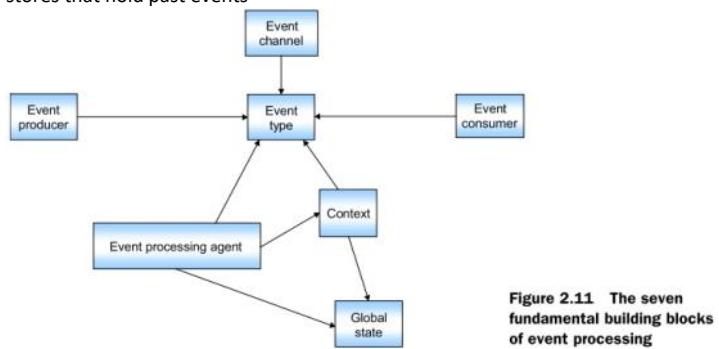


Figure 2.11 The seven fundamental building blocks of event processing

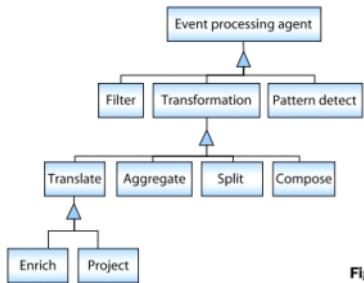


Figure 2.12 Different kinds of event processing agents

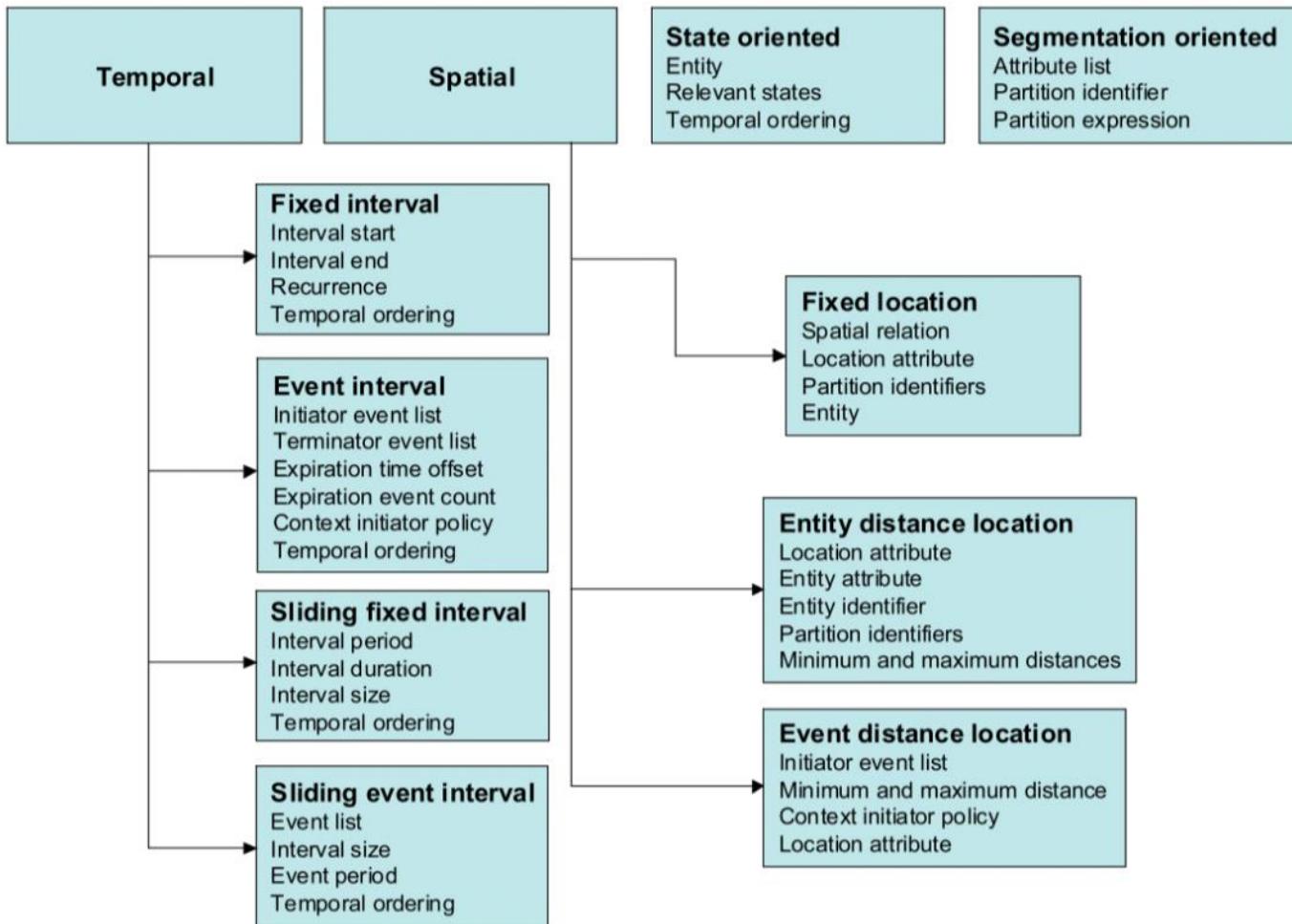


Figure 7.2 The different types of context showing their context parameters and partition parameters. This diagram also shows the principal dimension associated with a type, for example, the **fixed location** type is concerned with the spatial context dimension.

Single-event processing vs Multiple-event processing

- **Single-event processing**—A single event in the event stream will produce zero or more output data points or events.
- **Multiple-event processing**—Multiple events from the event stream will collectively produce zero or more output data points or events.

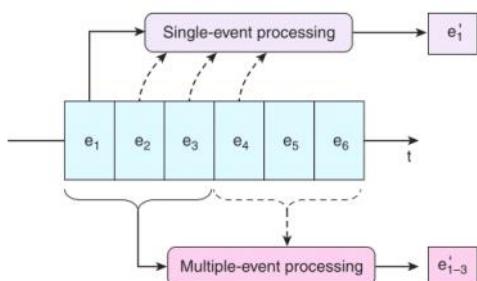


Figure 3.2 Our single-event processing app works on only one event from the source stream at a time. By contrast, the application at the bottom is reading three events from the source stream in order to generate a single output event.

(a) Single-event processing

- **Validating the event**—Checking, for example, “Does this event contain all the required fields?”

- *Validating* the event—Checking, for example, “Does this event contain all the required fields?”
- *Enriching* the event—Looking up, for example, “Where is this IP address located?”
- *Filtering* the event—Asking, for example, “Is this error critical?”

memory of a goldfish

- Независимо от того, какие преобразования мы пытаемся сделать, в случае одного события любая потоковая обработка концептуально проста, потому что мы должны действовать только с одним событием за раз. Наше приложение для потоковой обработки может иметь память золотой рыбки: не имеет значения ни одно событие, кроме того, которое читается прямо сейчас.

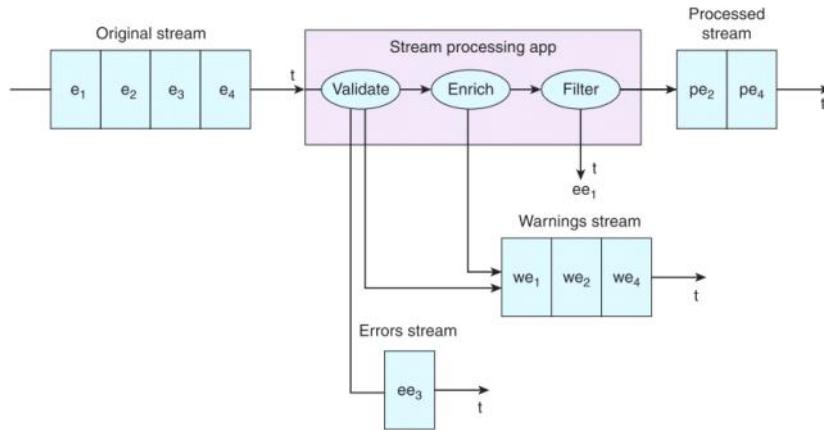


Figure 3.3 Here the stream processing application is validating, enriching, and filtering an incoming raw stream. Events that make it through the whole transformation are added to our processed stream. Events that fail validation are added to our errors stream. Transformation warnings are added to our warnings stream.

(b) Multiple-event processing

- Одновременная обработка нескольких событий значительно сложнее концептуально и технически, чем обработка отдельных событий
- *Aggregating*—Applying aggregate functions such as minimum, maximum, sum, count, or average on multiple events
- *Pattern matching*—Looking for patterns or otherwise summarizing the sequence, co-occurrence, or frequency of multiple events
- *Sorting*—Reordering events based on a sort key

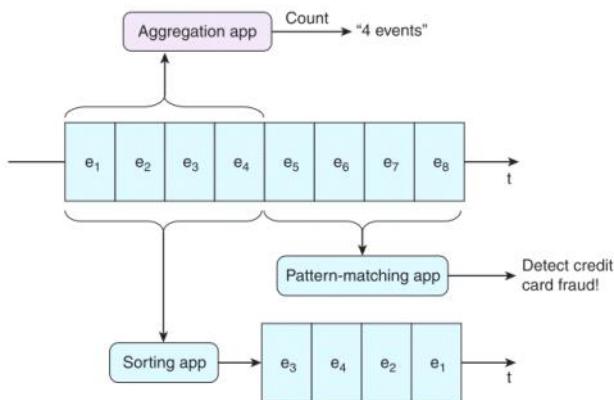


Figure 3.4 We have three apps processing multiple events at a time; an Aggregation app, which is counting events; a Pattern Matching app, which is looking for event patterns indicative of credit card fraud; and, finally, a Sorting app, which is reordering our event stream based on a property of each event.

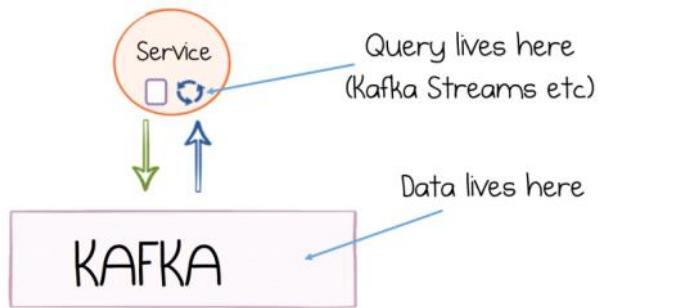
* materialized view

23 февраля 2021 г. 13:15

kafka - kind of distributed database: a **database 'unbundled'** (так бы сама база в центре a query engine много вокруг) (идея Jay Kreps)

- База данных - это действительно несколько концепций, объединенных в одну: хранилище, индексирование, кеширование, API запросов и иногда транзакционность, чтобы связать все это воедино. **Unbundling** - это идея, что вы можете разделить эти различные проблемы на разные уровни, и в этом есть преимущества.
- если вы немного прищуритесь, вы можете увидеть все системы и потоки данных вашей организации как единую распределенную базу данных. Вы можете просматривать все отдельные query-oriented systems (Redis, SOLR, Hive tables, and so on), **просто как индексы ваших данных**. Я заметил, что классическим специалистам по базам данных очень нравится это представление, потому что оно наконец объясняет им, что люди делают со всеми этими различными системами данных - это просто разные типы индексов!
- В основе unbundling лежит простая идея. Вместо того, чтобы использовать базу данных, в которой вы собираетесь в одном месте для всех ваших потребностей в данных, часто предпочтительнее разбить базу данных на части, а затем **составить решение из ее составных частей, каждая из которых точно настроена для выполнения определенной работы**. Причина, по которой это привлекательно, заключается в том, что системы, которые мы создаем, сами по себе очень похожи на эту: составленные отдельно друг от друга службы, как правило, хорошо выполняют одну задачу и образуют отдельные части более целостной головоломки.

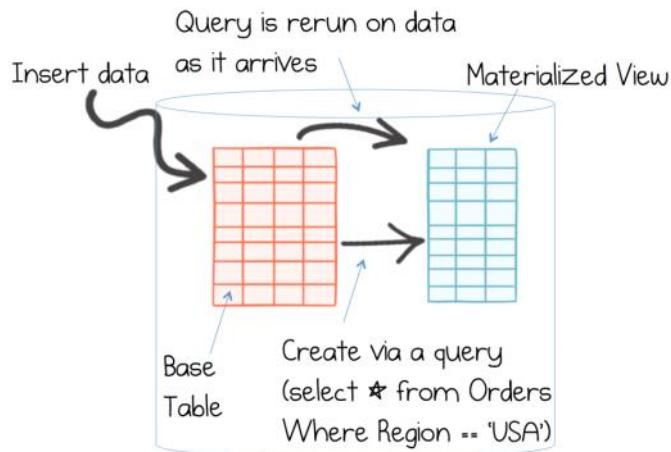
Data Storage + Query Engine == Database?



концепция **Materialized Views** (которые находятся отдельно от кафки и непрерывно обновляются сообщениями от кафки)

- В базе данных материализованное представление - это запрос, который база данных выполняет, а затем кэширует. При изменении каких-либо базовых данных обновляется и материализованное представление. Специалисты по базам данных используют их для оптимизации производительности чтения. Это работает, потому что представление предварительно вычисляет ваш запрос, чтобы получить данные в точной форме для вашего варианта использования. Поэтому, когда дело доходит до запроса представления, вся тяжелая работа уже сделана заранее.
- Проблема в том, что материализованное представление существует только внутри базы данных. Но если бы вы могли извлечь эту концепцию из базы данных в свои службы, у вас был бы постоянно обновляемый кеш (**continuously updated cache**). Кеши широко используются в современных приложениях, так что это кажется мощной идеей. Именно это и делают streaming platform.
- Децентрализация означает, что **представления можно размещать где угодно**. Они могут быть отдельным объектом. Они могут быть встроены в службу. Представление может принимать любую форму, которую вы хотите, и может быть восстановлен по желанию.
- Я вижу материализованное представление почти как своего рода кеш, который волшебным образом поддерживает себя в актуальном состоянии. Вместо того чтобы помещать всю сложность аннулирования кеша в приложение (рискуя условиями гонки и всеми обсуждаемыми проблемами), материализованные представления говорят, что обслуживание кеша должно быть ответственностью инфраструктуры данных.
- Решение состоит в том, чтобы построить материализованные представления из записей в журнале транзакций. Материализованные представления похожи на вторичные индексы, о которых мы говорили ранее: структуры данных, полученные из данных в журнале и оптимизированные для быстрого чтения. Материализованное представление - это всего лишь кэшированное подмножество журнала, и вы можете восстановить его из журнала в любое время.
- (b) Работа по созданию представления выполняется заранее, поэтому по умолчанию **оно будет оптимизировано для чтения**, материализованное представление вычисляется заранее, то есть все его содержимое вычисляется до того, как кто-либо его запросит
- (c) **Представление встроено прямо в нашу службу, поэтому оно работает быстро и локально**.

- (d) Основание вашей системы на журнале, а не на изменяемой базе данных, **позволяет избежать непредсказуемости или концентрации проблем согласованности**, которые связаны с общим изменяемым состоянием.
- Но в потоковой модели, **вместо того, чтобы периодически опрашивать базу данных и затем кэшировать результат**, мы должны определить представление, которое представляет точный набор данных, необходимый в прокручиваемой сетке, и потоковый процессор позаботится о его материализации за нас. Таким образом, вместо того, чтобы запрашивать данные в базе данных, а затем накладывать кэширование поверх, мы явно отправляем данные туда, где они необходимы, и обрабатываем их там



- To unbundle a materialised view into a continuously updating cache we need three separate things:
 - a mechanism for writing transactionally,
 - a log that maintains an immutable journal of these writes, and
 - a query engine that turns the journal into an index or view.

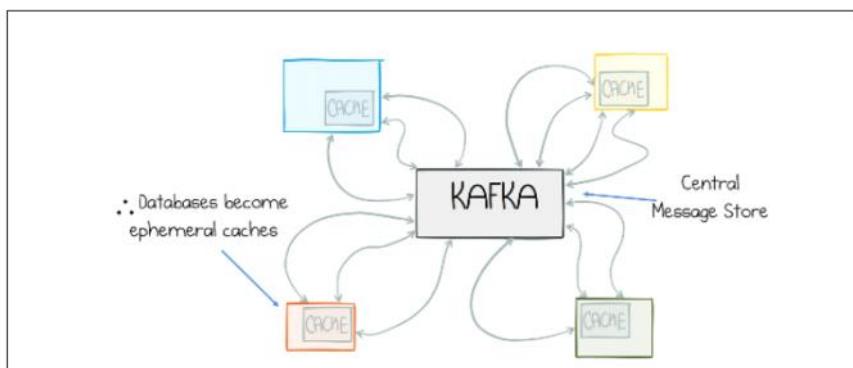
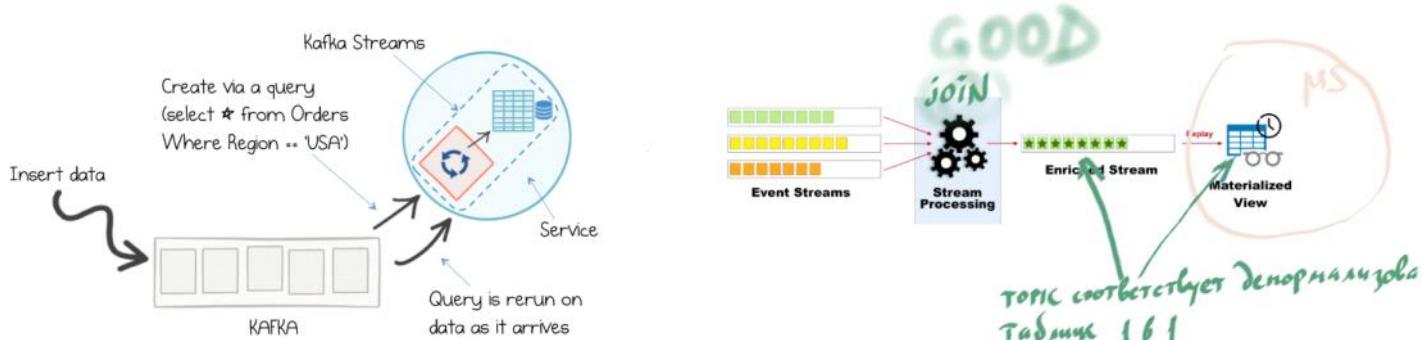


Figure 10-1. If the messaging system can store data, then the views or databases it feeds don't have to

Queryable state

- Чтение данных из changelog-топиков можно считать разновидностью материализованных представлений (materialized views).
- Для наших задач можно использовать определение материализованного представления из «Википедии»:

«...физический объект базы данных, содержащий результаты выполнения запроса. Например, оно может быть локальной копией удаленных данных, или подмножеством строк и/или столбцов таблицы или результатов соединения, или сводной таблицей, полученной с помощью агрегирования»

- Kafka Streams также позволяет выполнять интерактивные запросы (**interactive queries**) к хранилищам состояния, что дает возможность непосредственного чтения этих материализованных представлений.
Важно отметить, что запрос к хранилищу состояния носит характер операции «только для чтения». Благодаря этому вы можете не бояться случайно сделать состояние несогласованным во время обработки данных приложением.
- Возможность непосредственных запросов к хранилищам состояния имеет большое значение. Она значит, что можно создавать приложения — информационные панели без необходимости сначала получать данные от потребителя Kafka
 - благодаря локальности данных к ним можно быстро обратиться
 - исключается дублирование данных, поскольку они не записываются во внешнее хранилище
- можно напрямую выполнять запросы к состоянию из приложения. Нельзя переоценить возможности, которые это вам дает. Вместо того чтобы потреблять данные из Kafka и сохранять записи в базе данных для приложения, можно выполнять запросы к хранилищам состояния с тем же результатом. Непосредственные запросы к хранилищам состояния означают меньший объем кода (отсутствие потребителя) и меньше программного обеспечения (отсутствие потребности в таблице базы данных для хранения результатов).

interactive query = materialized view

- Идея интерактивных запросов не нова; аналогичная концепция фактически возникла в традиционных базах данных, где ее часто называют «материализованными представлениями»
- Благодаря нашему опыту работы с базами данных в сочетании с нашим опытом потоковой обработки мы рассмотрели ключевой вопрос: можно ли применить концепцию материализованных представлений к современному механизму потоковой обработки для создания мощной и универсальной конструкции для создания приложений и микросервисов с отслеживанием состояния?
- наша концепция заключалась в том, чтобы вывести потоковую обработку из ниши больших данных и сделать ее доступной в качестве основной модели разработки приложений.
- Для нас ключом к реализации этого видения является радикальное упрощение того, как пользователи могут обрабатывать данные в любом масштабе - малом, среднем, большом - и, по сути, одна из наших мантр - «Создавайте приложения, а не кластеры!»
- Интерактивные запросы позволяют разработчикам запрашивать встроенные хранилища состояний потокового приложения. Хранилища состояний хранят последние результаты узла процессора, например узла, который агрегирует данные для вычисления суммы или среднего



mutable state отделяется от immutable event log

- Отделяя изменяемое состояние от журнала неизменяемых событий, из одного и того же журнала событий можно получить разные представления, рассчитанные на чтение.
- Здесь можно провести аналогию с несколькими потребителями одного потока

состояние приложения также является своего рода материализованным представлением

- **application state** is also a kind of materialized view

- Аналогичным образом в случае источников событий состояние приложения поддерживается с помощью журнала событий;
- здесь состояние приложения также является своего рода материализованным представлением.
- В отличие от сценариев потоковой аналитики здесь, как правило, недостаточно рассматривать только события, произошедшие в рамках некоего временного окна.
- Для построения материализованного представления потенциально потребуются все события за произвольный период времени, за исключением устаревших, которые могут быть отброшены путем уплотнения журнала (см. пункт «Уплотнение журнала» подраздела «Перехват изменений данных» раздела 11.2).
- По сути, нам нужно окно, простирающееся до самого начала регистрации событий
- Samza и Kafka Streams поддерживают этот вид применения, опираясь на реализованные в Kafka возможности сжатия журнала

основным источником coupling являются сами данные в кафке (low-coupling)

- Unlike a database, a log provides a low-coupling mechanism for sharing datasets. The loose coupling comes from the simple interface the log provides: little more than seek and scan, meaning the dominant source of coupling is the data itself.

query layer поверх лога кафки

- Когда слой запроса добавляется поверх журнала, скажем, с использованием потоков Kafka, **микросервисы сохраняют контроль над тем, какие запросы выполняются, на каком оборудовании они выполняются, какие преобразования применяются для создания представлений** (так например можно RDBMS а можно Elasticsearch) и, что наиболее важно: когда и как эти вещи изменены.
- По сравнению с традиционной централизованной базой данных **именно этот уровень контроля обеспечивает маневренность и гибкость сервисов**.
- Когда микросервис необходио зарелизить, все компоненты с высокой степенью coupling уже находятся внутри ее развертываемого модуля. Разделяются только сами данные.
- Таким образом, несвязанная база данных имеет функциональные свойства, аналогичные обычной базе данных, но без всех плохих частей.
- Ключевой вывод по сути тот же: потоковая обработка отделяет ответственность **механизма хранения данных от механизма, используемого для их запроса**. Таким образом, может быть один общий поток событий, скажем, для платежей, но много специализированных представлений в разных частях компании (рис. 9-2). Но это обеспечивает важную альтернативу традиционным механизмам интеграции данных.

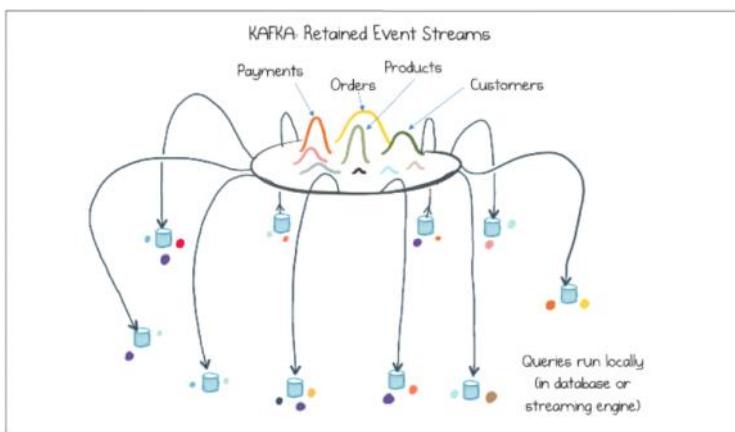


Figure 9-2. A number of different applications and services, each with its own views, derived from the company's core datasets held in Kafka; the views can be optimized for each use case

у каждого микросервиса свой query

- Query functionality **is not shared**; it is private to each service, allowing teams to move quickly by retaining control of the datasets they use.
- Как мы увидим в главе 10, это приводит к некоторой дальнейшей оптимизации, при которой каждое представление оптимизируется для конкретного случая использования, почти так же, как материальные представления используются в реляционных базах данных для создания оптимизированных для чтения, использования наборы данных, ориентированные на случаи.

serving layer

- Система разделена на две логические части: журнал и обслуживающий слой. Журнал фиксирует изменения состояния в последовательном порядке. Обслуживающие узлы хранят любой индекс, необходимый для обслуживания запросов (например, в хранилище значений ключей может быть что-то вроде btree или sstable, а у поисковой системы будет инвертированный индекс)
- Обслуживающие узлы подписываются на журнал и как можно быстрее применяют записи к его локальному индексу в том порядке, в котором они были сохранены в журнале.

separate the query characteristics from the availability and consistency aspects of the system

- Я считаю, что разделение на query API и лог, очень показателен, поскольку позволяет отделить характеристики запроса от аспектов доступности и согласованности системы. Я на самом деле думаю, что это даже полезный способ мысленно проанализировать систему, которая не построена таким образом, чтобы лучше ее понять.

преимущества инкрементальной разработки: я могу сделать новую версию представления, даже если она полностью несовместима со старой версией

(это возможно благодаря автовосстановлению вьюхи из кафки, т.e я могу полностью выкинуть старую

вьюху и построить вьюху с абсолютно новой структурой)

(или можно даже сделать две вьюхи рядом в целях тестирования)

- Одно интересное следствие этого стиля системы состоит в том, что представления не обязательно должны быть долгоживущими. Если вы возьмете копию каталога продуктов и сопоставите ее с моделью внутренней предметной области, в случае изменения этой модели вы можете просто выбросить представление и перестроить его.
- Когда у вас есть этот процесс для преобразования журналов в представления, у вас появляется большая гибкость для создания новых представлений: если вы хотите представить свои существующие данные каким-либо новым способом, вы можете просто создать новое задание обработки потока, использовать журнал ввода с самого начала, и таким образом построить совершенно новый взгляд на все существующие данные. Затем вы можете поддерживать оба представления параллельно, постепенно перемещать приложения в новое представление и в конечном итоге отбрасывать старый вид. Больше никаких пугающих миграций типа "остановки мира".
- Что мне особенно нравится, так это то, что эта архитектура помогает обеспечить гибкую инкрементную разработку программного обеспечения. Если вы хотите поэкспериментировать с новой функцией продукта, для которой вам нужно представить существующие данные по-новому, вы можете просто создать новое представление для своих данных, не затрагивая ни одно из существующих представлений. Затем вы можете показать это представление подмножеству пользователей и проверить, лучше ли оно старого. Если да, вы можете постепенно переводить пользователей на новое представление; в противном случае вы можете просто отказаться от просмотра, как будто ничего не произошло. Это намного более гибко, чем миграция схемы, которая обычно выполняется по принципу «все или ничего». Возможность свободно экспериментировать с новыми функциями без обременительных процессов миграции - это потрясающий инструмент.

Сервисам рекомендуется принимать только те данные, которые им нужны, в определенный момент времени (те типы данных впрок не запасать).

т.e изイベントов забирать только нужную информацию и отфильтровывать ненужное, чтобы stateful store

был маленьким

т.e конкретная вьюха сделана только под конкретную задачу

- Это делает виды маленькими и легкими. Это также снижает сцепление
- это отличается от классических мега БД, которые содержат все
- Наконец, простота создания представлений на платформе потоковой передачи означает, что вы можете создавать больше из них, специально предназначенных для конкретной задачи. Так что, развивая этот стиль системы, вы обнаруживаете, что создаете множество специфичных для службы представлений. Резкий контраст с направлением всех запросов в единую централизованную базу данных или службу данных.
- Служба инвентаризации, обсуждаемая в главе 15 , является хорошим примером. Он считывает сообщения инвентаризации, которые включают много информации о различных продуктах, хранящихся на складе. Когда сервис читает каждое сообщение, он отбрасывает почти все в документе, беря только два поля: идентификатор продукта и количеству товаров на складе.
- Reducing the breadth of the view keeps the dataset small and loosely coupled. By keeping the dataset small you can store more rows in the available memory or disk, and performance will typically improve as a result. Coupling is reduced too, since should the schema change, it is less likely that the service will store affected fields.

scale a materialised view out horizontally (local state store как полный аналог одной шарды в шардированной БД)

Также возможно масштабировать материализованное представление по горизонтали, если данных слишком много или если пропускная способность сети слишком высока для одного сервера. При работе в распределенной конфигурации Kafka Streams предоставляет информацию об обнаружении, необходимую для маршрутизации запросов ключа на соответствующий сервер.

* внешние и внутренние данные

23 февраля 2021 г. 13:19

бизнес-факты выпускаемые одним микросервисом это **внешние данные** потребляемые другим микросервисом (тк микросервисы не могут напрямую ходить друг к другу в базу) (концепция внешних данных Пэта Хэлланда)

- Общий источник истины оказывается на удивление полезным. Например, микросервисы не делятся своими базами данных друг с другом (это называется антипаттерном **IntegrationDatabase**). Для этого есть веская причина: базы данных имеют очень богатые API-интерфейсы, которые удивительно полезны сами по себе, но когда они широко используются, они затрудняют решение, будет ли и как одно приложение повлиять на другие, будь то связывание данных, конкуренция или нагрузка. Но **бизнес-факты, которыми службы предпочитают делиться, являются наиболее важными фактами из всех.** Это правда, на которой построен весь остальной бизнес. Пэт Хелланд назвал это различие еще в 2006 году, обозначив его «**внешние данные**».
- A shared source of truth turns out to be a surprisingly useful thing. Microservices, for example, don't share their databases with one another (referred to as the **IntegrationDatabase antipattern**). There is a good reason for this: databases have very rich APIs that are wonderfully useful on their own, but when widely shared they make it hard to work out if and how one application is going to affect others, be it data couplings, contention, or load. But the business facts that services do choose to share are the most important facts of all. They are the truth that the rest of the business is built on. Pat Helland called out this distinction back in 2006, denoting it "**data on the outside**".
- Но воспроизводимый журнал предоставляет гораздо более подходящее место для хранения таких данных, потому что (как ни странно) вы не можете запросить его! Речь идет исключительно о хранении данных и отправке их в новое место. Идея чистого перемещения данных важна, потому что **внешние данные - общие для служб данных - являются наиболее тесно связанными из всех, и чем больше услуг имеет экосистема, тем более тесно связаны эти данные.** Решение состоит в том, чтобы **переместить данные куда-нибудь с более слабой связью**, то есть переместить их в ваше приложение, в котором вы можете манипулировать им сколько душе угодно. Таким образом, перемещение данных дает приложениям уровень работоспособности и контроля, недостижимый при прямой зависимости от времени выполнения. Эта идея сохранения контроля оказывается важной - по той же причине, по которой шаблон совместно используемой базы данных не работает на практике.
- But a replayable log provides a far more suitable place to hold this kind of data because (somewhat counterintuitively) you can't query it! It is purely about storing data and pushing it to somewhere new. This idea of pure data movement is important, because data on the outside—the data services share—is the most tightly coupled of all, and the more services an ecosystem has, the more tightly coupled this data gets. The solution is to move data somewhere that is more loosely coupled, so that means moving it into your application where you can manipulate it to your heart's content. So data movement gives applications a level of operability and control that is unachievable with a direct, runtime dependency. This idea of retaining control turns out to be important—it's the same reason the shared database pattern doesn't work out well in practice
- **Data on the Outside** refers to the information that flows between these independent services. Since we defined services as being connected by messaging, it is reasonable to consider all outside data as being transmitted with a message

	Outside Data	Inside Data
Immutable?	Yes	No
Identity-Based References	Yes	No
Open Schema?	Yes	No
Represent in XML?	Yes	No
Encapsulation Useful?	No	Yes
Long-Lived Evolving Data with Evolving Schema?	No	Yes
Business Intelligence Desirable over Data?	Yes	Yes
Durable Storage in SQL Inside the Service?	Yes: Copy of XML Kept in SQL	Yes

с внутренними данными (которые получены из внешних) микросервис может делать все что угодно, у него максимальная гибкость

- становится ясно, что данные за пределами службы данных - общие для служб данных - необходимо тщательно курировать и развивать, но чтобы сохранить нашу свободу повторения, нам необходимо превратить их в данные внутри, чтобы мы могли сделать это своим.
- те выполняется основной принцип Centralize an immutable stream of facts Decentralize the freedom to act, adapt, and change.
- Data on the Inside refers to the encapsulated private data contained within the service itself. As a sweeping statement, this is the data that we have always considered as "normal data". The classic data contained in a SQL database and manipulated by a typical application is inside data.
- Как описано выше, внутренние данные инкапсулируются за логикой приложения службы. Это означает, что единственный способ изменить данные - через логику приложения службы. Услуги предлагают очень прочную и жесткую герметизацию. Основное понятие заключается в том, что к базовым данным нет никакого доступа, если только он не опосредован логикой приложения службы. Внутренняя часть службы не видна.

данные - это first class citizen

- внешние данные - доля служб данных - сами по себе становятся важной сущностью
- (1) по мере роста архитектур и становления систем более ориентированными на данные, перемещение наборов данных от сервиса к сервису становится неизбежной частью эволюции систем;
- (2) внешние данные - доля служб данных - сами по себе становятся важной сущностью;
- (3) совместное использование базы данных не является разумным решением для данных извне, но совместное использование воспроизводимого журнала лучше уравновешивает эти проблемы, поскольку оно может хранить наборы данных в долгосрочной перспективе и облегчает программирование, управляемое событиями, реагируя на настоящее.
- Такой подход позволяет синхронизировать данные многих сервисов через слабо связанный интерфейс, давая им свободу локально нарезать, нарезать кубиками, обогащать и развивать данные.

* Database Inside Out

23 февраля 2021 г. 13:21

поток данных всей организации начинает выглядеть как одна огромная БД

- Всякий раз, когда пакет, поток или ETL-процесс переносит данные из одних места и формы в другие место и форму, он действует подобно подсистеме базы, которая постоянно обновляет индексы и материализованные представления
- Как уже обсуждалось, подобные вещи происходят внутри базы, когда триггер реагирует на изменение данных или обновляется вторичный индекс, отражая изменение индексируемой таблицы. Разделение БД и приложений означает принятие этой идеи и применение ее к созданию производных наборов данных за пределами первичной базы: кэши, полнотекстовые поисковые индексы, машинное обучение или аналитические системы. Для этой цели можно использовать потоковую обработку и системы обмена сообщениями.

С этой точки зрения пакетные и потоковые процессоры подобны сложным триггерам, хранимым процедурам и процедурам обслуживания материализованных представлений.

- Поддерживаемые ими производные системы данных подобны индексам разных типов. Например, реляционная БД может поддерживать индексы Б-деревьев, хеш- и пространственные индексы (см. пункт «Составные индексы» подраздела «Другие индексные структуры» раздела 3.1) и другие их типы. В новой архитектуре производных информационных систем эти возможности не реализованы как функции единой интегрированной базы данных, а предоставляются различными программными продуктами, работающими на разных машинах и администрируемыми разными группами людей

“A Database Inside Out” база-данных наизнанку (те сначала пишем в лог а потом уже обновляем саму БД ее индексы и тп) (БД вывернутая наизнанку - концепция Мартина Клепмана)

- некоторым нравится сравнивать Kafka с базой данных. Он, безусловно, имеет аналогичные функции. Он обеспечивает хранение; производственные темы с сотнями терабайт не редкость. Он имеет интерфейс SQL, который позволяет пользователям определять запросы и выполнять их над данными, хранящимися в журнале. Их можно передать по конвейеру в представления, которые пользователи могут запрашивать напрямую. Он также поддерживает транзакции. Это все вещи, которые по своей природе кажутся довольно «databasely»
- Kafka is a database inside out (see “A Database Inside Out” on page 87 in Chapter 9), a tool for storing data, processing it in real time, and creating materialized views.
- По сути, это идея, что база данных имеет ряд основных компонентов - журнал фиксации, механизм запросов, индексы и кеширование - и вместо того, чтобы объединять эти проблемы в рамках единой технологии черного ящика, как это делает база данных, мы можем разделить их на отдельные части с помощью инструментов потоковой обработки, и эти части могут существовать в разных местах, соединенных вместе журналом. Таким образом, Kafka играет роль журнала фиксации, потоковый процессор, такой как Kafka Streams, используется для создания индексов или представлений, и эти представления ведут себя как форма постоянно обновляемого кеша , находящегося внутри вашего приложения или рядом с ним (рис. 9-1)
- мы глубоко укоренились в том, что внедрять бизнес-логику в базу данных - плохая идея. Но обратное - добавление данных в ваш код - открывает множество возможностей для объединения наших данных и нашего кода.
- «База данных наизнанку» - это аналогия потоковой обработки, когда те же компоненты, которые мы находим в базе данных - журнал фиксации, представления, индексы, кэши - не ограничены одним местом, а вместо этого могут быть доступны везде, где они необходимы.
- Использование лог-журнала в базах данных связано с синхронизацией различных структур

данных и индексов при сбоях. Чтобы сделать это атомарным и надежным, база данных использует журнал для записи информации о записях, которые они будут изменять, прежде чем применять изменения ко всем различным структурам данных, которые она поддерживает. Журнал - это запись того, что произошло, и каждая таблица или индекс - это проекция этой истории в некую полезную структуру данных или индекс. Поскольку журнал сохраняется сразу, он используется как авторитетный источник для восстановления всех других постоянных структур в случае сбоя

метафора

- Даже в электронных таблицах есть возможности программирования потока данных, и они намного опережают большинство обычных языков программирования [33]. В электронной таблице можно ввести формулу в ячейку (например, вычисление суммы значений из ячеек другого столбца), и всякий раз, когда изменится одна из этих ячеек, результат формулы будет автоматически пересчитан. Именно это мы хотим получить на уровне информационной системы: при изменении записи в БД все индексы для данной записи должны автоматически обновляться. Кроме того, следует автоматически обновиться и всем кэшированным представлениям и агрегациям, зависящим от этой записи. Вы не должны думать о технических деталях происходящего, но притом должны быть уверены в правильной работе всех элементов-участников.

как происходит в обычных БД: [write-ahead log](#)

- Базы данных, реализуют идею журнала упреждающей записи .
- Вставки и обновления сразу же последовательно записываются на диск, как только они поступают. Это делает их долговечными,
- поэтому база данных может отвечать пользователю, зная, что данные безопасны, но не дожидаясь медленного процесса обновления различных параллельных структур данных, таких как таблицы, индексы и т.
- Дело в том, что (а) если что-то пойдет не так, внутреннее состояние базы данных может быть восстановлено из журнала и (б) операции записи и чтения могут быть оптимизированы независимо.

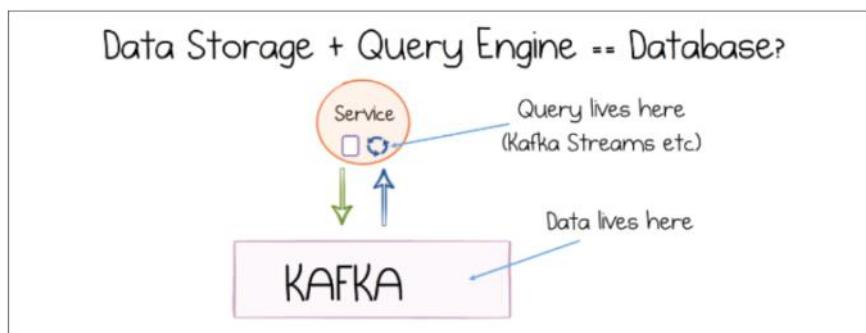


Figure 9-1. A streaming engine and a replayable log have the core components of a database

Что происходит, когда мы делаем запрос CREATE INDEX с целью создать новый индекс в реляционной БД?

- База проверяет согласованный снимок таблицы, выбирает все индексируемые значения полей, сортирует их и генерирует на выходе индекс.
- Затем она будет обрабатывать записи, сделанные с момента создания согласованного снимка (при условии, что при создании индекса таблица не была заблокирована, вследствие чего в ней могут появляться новые записи).
- После того как это будет сделано, база данных должна будет обновлять индекс всякий раз при выполнении записи в таблицу в ходе транзакции.

Этот процесс очень похож на создание копии ведомого узла , а также на настройку перехвата изменений данных в потоковой системе

в EDA схемы сообщений это контракт (а НЕ точки микросервисов как в WSDL, swagger, GraphQL)

- поэтому общий реестр схем это важно (особенно для runtime проверок)
- В отсутствие формальной интеграции между системами наши схемы событий являются наиболее близким к контракту между системой, генерирующей поток событий, и системами, потребляющими этот поток
- Производитель события соглашается схему события с первоначальным известным потребителем события. Это действует как договор между обеими сторонами

APIs between services are Contracts

In Event Driven World – Event Schemas ARE the API

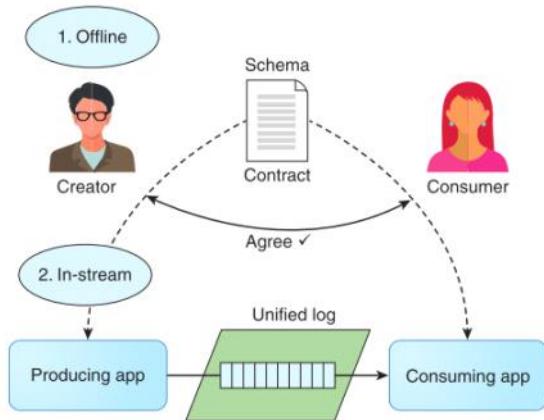
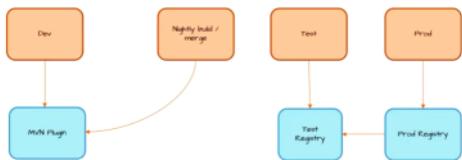


Figure 6.2 The producer of the event agrees to the schema of the event with the initial known consumer of the event. This acts as a contract between both parties. Then, in the stream, the producing app creates events using the agreed-upon schema, and the consuming app can happily read those events, safe in the knowledge that the events will conform to the schema.

но также схемы можно проверять еще compile time с помощью maven/gradle плагина

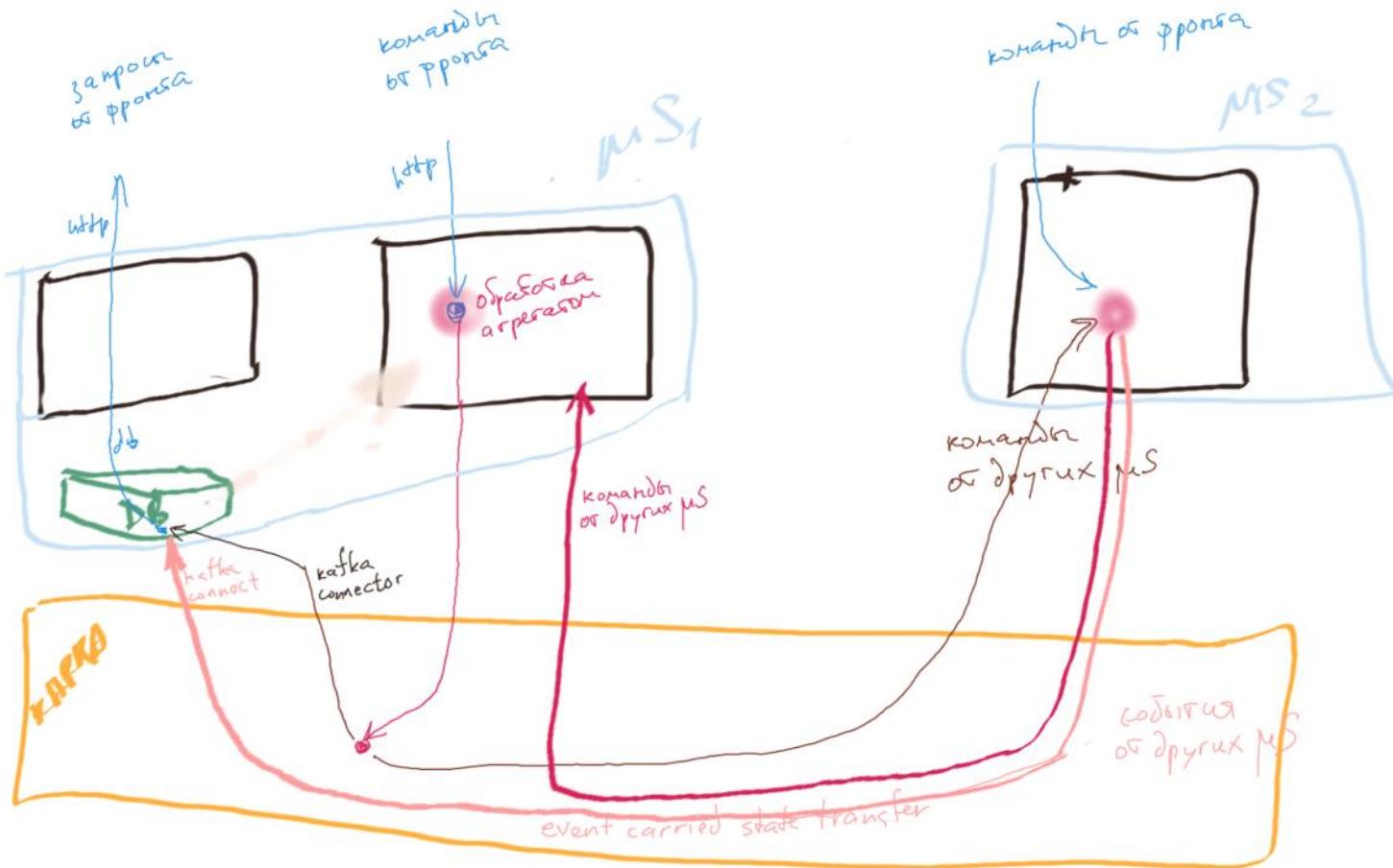
So the flow is...



Брокеры сообщений обычно не навязывают какой-либо конкретной модели данных — сообщение представляет собой просто байтовую последовательность с метаданными,

- так что можно использовать любой формат кодирования

??



Events: Schema First

1 model / event

N events in one union type

1 union type / stream

Stream owned by 1 service

Most services define exactly 1 stream

```

"unions": {
  "user_event": {
    "discriminator": "discriminator",
    "types": [
      { "type": "user_upserted" },
      { "type": "user_deleted" }
    ]
  }
},
"models": {
  "user_upserted": {
    "fields": [
      { "name": "event_id", "type": "string" },
      { "name": "timestamp", "type": "date-time-iso8601" },
      { "name": "user", "type": "io.flow.common.v0.models.user" }
    ]
  },
  "user_deleted": {
    "fields": [
      { "name": "event_id", "type": "string" },
      { "name": "timestamp", "type": "date-time-iso8601" },
      { "name": "user", "type": "io.flow.common.v0.models.user" }
    ]
  }
}

```

Producing an Event

```

override def process(record: UserVersion)(implicit ec: ExecutionContext): Unit = {
  record.journalOperation match {
    case ChangeType.Insert | ChangeType.Update => {
      stream.publish(
        UserUpserter(
          eventId = eventIdGenerator.randomUUID(),
          timestamp = DateTime.now,
          user = record.userVersion.user
        )
      )
    }
    case ChangeType.Delete => {
      stream.publish(
        UserDeleted(
          eventId = eventIdGenerator.randomUUID(),
          timestamp = DateTime.now,
          id = record.userVersion.user.id
        )
      )
    }
  }
}

```

Note the UserVersion class which is also code generated.

Guarantees that all code in all services looks the same.

EDA event driven architecture

- отличается от message driven architecture на основе очередей

https://en.wikipedia.org/wiki/Event-driven_architecture

contextual event driven application (те огромные data lake хранилища данных в стороне все же есть)

- те не просто event driven а с внешними данными те с контекстом
- real time events + historical/static/reference data
- speed layer и slow layer
- [onenote:MICROSERVICES%201.one#contextual%20event-driven%20applications§ion-id={7172B8D4-BDE4-4FFF-A011-59DE79779A94}&page-id={751AEFAE-681B-48A5-B449-DE246AB0862E}&end&base-path=C:\Users\trans\Qsync\nova_from_onenote\tf_алгоноте_v1\SYSTEMDESIGN\KAFKA](#)

(+) потоков style и functional style лучше всего подходят для distributed design

- Два стиля программирования, которые лучше подходят для распределенного проектирования, особенно в контексте сервисов, - это стили потока данных и функциональные стили.
- Этот стиль программы больше похож на сборочную линию, где каждый рабочий выполняет определенную задачу, пока продукты продвигаются по конвейерной ленте. Поскольку каждый работник озабочен только доступностью входных данных, у него нет «скрытого состояния» для отслеживания. Это очень похоже на то, как работают потоковые системы. События накапливаются в потоковом процессоре, ожидая выполнения условия, например, операции соединений.
- Потоковая передача имеет неотъемлемую способность к распараллеливанию.
- Потоковая передача, естественно, позволяет создавать кэшированные наборы данных и поддерживать их в актуальном состоянии. Это делает ее хорошо подходящим для систем, в которых данные и код разделены сетью, особенно для обработки данных и графических интерфейсов.
- т-ф: микросервис может быть однопоточным (то проще программировать)
- Близким родственником функциональной парадигмы является dataflow style (программирование потоков данных). Различие состоит в том, что первый фокусируется на функциях и их составе, тогда как второй концентрируется на перемещении данных через сеть (точнее, направленный ациклический граф) связанных вычислений

(+) functional style лучше всего подходят для distributed design

- Есть такая же полезная аналогия с функциональным программированием. Как и в случае со стилем потока данных, состояние не меняется на месте, а скорее эволюционирует от функции к функции, и это полностью соответствует способу работы потоковых процессоров . Таким образом, большинство преимуществ как функционального языка, так и языка потока данных также применимы к потоковым системам.
- Программирование в чисто функциональном стиле означает, что оценка каждого выражения всегда дает одинаковый результат при одинаковых входных данных . Побочных эффектов нет. Это позволяет компилятору планировать оценку по мере необходимости вместо того, чтобы делать это строго в порядке, указанном в исходном коде, включая возможность параллельного выполнения. Необходимым предварительным условием для этого является неизменность всех значений - ничто не может измениться после того, как было создано. Это имеет очень выгодное следствие, что значения могут свободно распределяться между одновременно выполняющимися потоками без какой-либо необходимости в синхронизации

microservices approach vs. dataflow approach

- Например, предположим, что покупатель приобретает товар, стоимость которого оценивается в одной валюте, но оплачивается в другой. Для выполнения конвертации валюты необходимо знать текущий обменный курс. Эта операция может быть реализована двумя способами

(+) microservices approach

- Методом микросервисов. Программный код, обрабатывающий покупки, вероятно, обратится к сервису обменных курсов или БД, чтобы получить текущий курс для данной валюты.

(+) dataflow approach

- Методом потока данных. Программный код, обрабатывающий покупки, заранее подпишется на поток обновлений обменного курса и при каждом изменении текущих значений будет записывать их в локальную базу данных. При необходимости обработать покупку ему нужно только сделать в нее запрос
- Во втором варианте вместо синхронного сетевого запроса другого сервиса использован запрос локальной базы данных (которая может находиться на том же компьютере, даже в том же процессе)1
- . Технология потоков данных не только быстрее, но и более устойчива к отказу другого сервиса. Самый быстрый и надежный сетевой запрос — это отсутствие сетевых запросов! Теперь вместо RPC у нас есть потоковое объединение между событиями купли-продажи и событиями обновления обменного курса (см. пункт «Объединения «поток — таблица»

* кафка - как messaging system

23 февраля 2021 г. 13:22

кафка - как messaging system НО с бессрочным хранением сообщений

- система обмена сообщениями, **оптимизированная для хранения** наборов данных, будет более подходящей, чем база данных, оптимизированная для их публикации.
- приложений и сервисов, интегрированных посредством событий, но также использовавших историческую справку, которую они могли бы использовать. бизнес-операции по-прежнему нуждаются в **исторических данных** - будь то пользователи, желающие запросить историю своих учетных записей, некоторые службы, которым требуется список клиентов, или аналитика, которую необходимо запустить для управленческого отчета..
- Но идея перестройки системы вокруг событий не нова - архитектуры, управляемые событиями, существуют уже несколько десятилетий, а такие технологии, как корпоративный обмен сообщениями, являются крупным бизнесом, особенно (что неудивительно) для корпоративных компаний.

одновременно можно реагировать на реалтайм события или отправить группу ситорических фактов

- Таким образом, этот подход на основе повторяемого журнала имеет два основных преимущества. Во-первых, он позволяет легко реагировать на события, которые происходят сейчас, с помощью набора инструментов, специально разработанного для управления ими. Во-вторых, он предоставляет центральный репозиторий, который может отправлять целые наборы данных туда, где они могут быть необходимы. Это очень полезно, если вы ведете глобальный бизнес с центрами обработки данных, разбросанными по всему миру, вам нужно быстро загрузить или создать прототип нового проекта, провести разовое исследование данных или создать сложную экосистему услуг, которая может развиваться свободно и независимо.

Kafka is a message broker

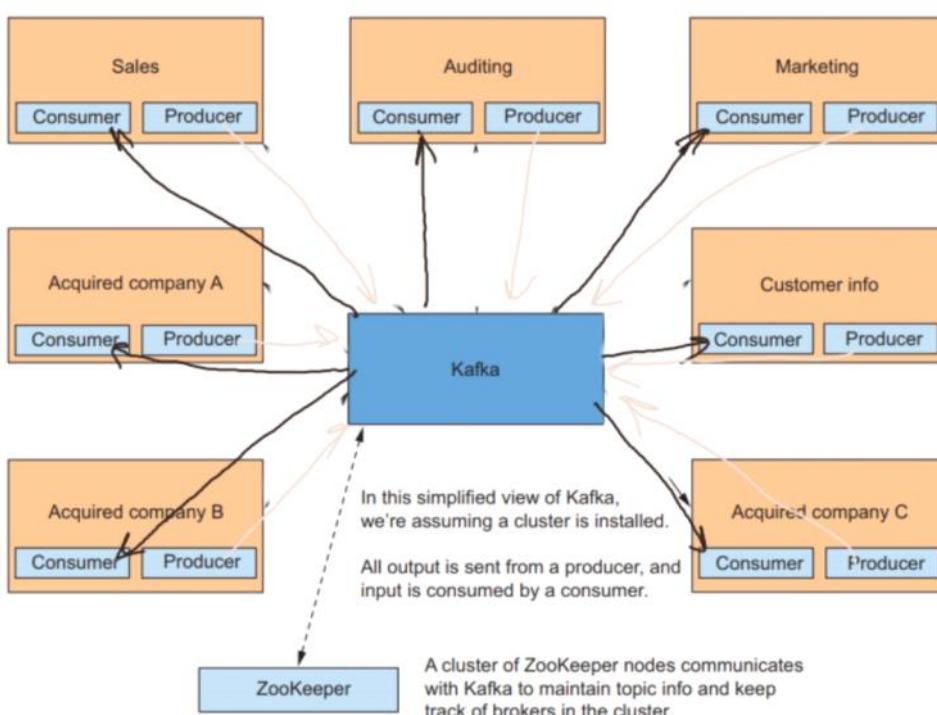


Figure 2.4 Kafka is a message broker. Producers send messages to Kafka, and those messages are stored and made available to consumers via subscriptions to topics.

заинтересованные в мгновенном получении события могут их сразу получить
оперативность единого журнала - одна из его самых мощных функций

- но сразу не получат тк ивенты сначала должны быть закомичены в логе

кафка это it's not a data processing framework but instead a transport layer

- persistent streaming transport, implemented as a set of partitioned logs

- Kafka сыграл одну из самых влиятельных ролей в продвижении потоковой обработки из всех систем
- До Kafka в большинстве систем потоковой обработки для отправки данных использовались какие-то эфемерные системы очередей, такие как Rabbit MQ, или даже старые TCP-сокеты. Долговечность может быть в некоторой степени обеспечена за счет резервного копирования в восходящем направлении в производителях (т. Е. Возможность для вышестоящих производителей данных повторно отправлять данные, если последующие рабочие вышли из строя), но часто восходящие данные также хранились эфемерно. И большинство подходов полностью игнорировали идею возможности повторного воспроизведения входных данных позже
- Кафка все изменил. Взяв закаленную в боях концепцию долговечного журнала из мира баз данных и применив ее к области потоковой обработки, Kafka вернул нам всем чувство безопасности, которое мы потеряли при переходе от надежных источников ввода, распространенных в Hadoop / пакетная обработка эфемерных источников, преобладающих в то время в мире потоковой передачи.

replayability - Повторяемость - это основа, на которой построены сквозные гарантии exactly-once точного однократного выполнения в Apex, Flink, Kafka Streams, Spark и Storm.

- При выполнении в одноразовом режиме каждая из этих систем предполагает / требует, чтобы источник входных данных имел возможность перематывать и воспроизводить все данные до самой последней контрольной точки. При использовании с источником ввода, который не обеспечивает такой возможности (даже если источник может гарантировать надежную доставку через резервное копирование восходящего потока), сквозная семантика «ровно один раз exactly-once» разваливается.

различие между брокером и БД (см [прямая и непрямая отправка](#))

- Информация в БД обычно хранится до тех пор, пока не будет явно удалена, тогда как большинство брокеров автоматически удаляют сообщение после того, как оно было успешно доставлено потребителям
- предполагается, что очереди сравнительно коротки. Если из-за медленных потребителей брокеру приходится буферизовать много сообщений, то на обработку каждого сообщения уходит больше времени, а общая пропускная способность может снизиться
- Базы обычно поддерживают вторичные индексы и различные способы поиска данных, в то время как брокеры сообщений обеспечивают какой-либо способ подписки на подмножество тем, соответствующих определенному шаблону
- Результат запроса в БД обычно основан на ее текущем состоянии в данный момент времени; если другой клиент впоследствии что-то запишет в базу и это изменит результат запроса, то первый клиент не узнает об устаревании его предыдущего результата (при условии, что не повторяет запрос или не будет опрашивать базу об изменениях). Брокеры сообщений, напротив, не поддерживают произвольные запросы, но уведомляют клиентов об изменении данных (то есть о появлении новых сообщений).

различие между очередью и логом-кафки

- В этом кроется отличие от систем обмена сообщениями: очереди по умолчанию сохраняют сообщения в памяти и записывают их на диск только в том случае, если очередь становится слишком длинной. Такие системы бывают быстрыми для коротких очередей и становятся намного медленнее, когда начинают записывать данные на диск, их скорость зависит от объема сохраненной истории

- у традиционных брокеров сообщений, где нужно тщательно вручную удалять запросы, чьи потребители были отключены, — в противном случае они продолжают напрасно накапливать сообщения и отнимать память у потребителей, которые все еще активны
- а кафке через неделю сообщения сами чистятся
- в брокерах сообщений типа AMQP и JMS обработка и подтверждение сообщений является деструктивной операцией, поскольку в результате брокер удаляет эти сообщения.
- В kafka-брокере сообщений на основе журнализации, наоборот, обработка больше похожа на чтение из файла: это операция только чтения, не меняющая журнал.

* кафка как хранилище

23 февраля 2021 г. 13:24

кафка как хранилище

storage layer - кафку можно использовать как хранилище (но без богатого query language)

Одно из самых больших различий между Kafka и другими системами обмена сообщениями заключается в том, что его можно использовать в качестве уровня хранения . На самом деле, нередко можно увидеть темы, основанные на удержании или сжатые, содержащие более 100 ТБ данных. Но Kafka - это не база данных; это журнал фиксации, не предлагающий широкой функциональности запросов (и мы не планируем его менять). Но его простой контракт оказывается весьма полезным для хранения общих наборов данных в больших системах или архитектурах компании - например, для использования событий в качестве общего источника истины

Данные могут храниться в обычных темах, которые отлично подходят для аудита или поиска событий, или в сжатых темах, которые уменьшают общую площадь. Вы можете объединить эти два аспекта, в паттерне **latest-versioned pattern**

В течение нескольких десятилетий системы обмена сообщениями использовали эти свойства, перемещая события от системы к системе, но только в последние несколько лет системы обмена сообщениями начали использоватьсь в качестве уровня хранения, сохраняя наборы данных, которые проходят через них.. Это создает интересный архитектурный образец. Основные наборы данных компании хранятся в виде централизованных потоков событий со всеми эффектами развязки встроенного брокера сообщений. Но в отличие от традиционных брокеров, которые удаляют сообщения после того, как они были прочитаны, исторические данные хранятся и доступны любой команде, которая в них нуждается. . Это тесно связано с идеями, разработанными в Event Sourcing (см. Главу 7), и с концепцией внешних данных Пэта Хелланда . ThoughtWorks называет этот шаблон потоковой передачей событий источником истины .

[кафка как концентратор информации](#)

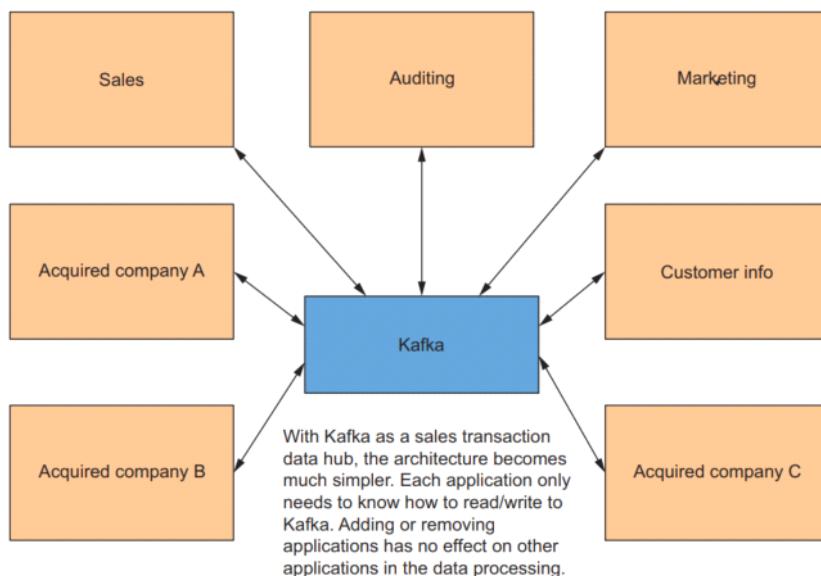


Figure 2.3 Using Kafka as a sales transaction hub simplifies the ZMart data architecture significantly. Now each machine doesn't need to know about every other source of information. All they need to know is how to read from and write to Kafka.

[kind of event stream database](#)

Журнал действует как очень, очень большой буфер, который позволяет перезапустить процесс или выйти из строя, не замедляя другие части графа обработки

- Во-вторых, журнал обеспечивает буферизацию процессов. Это очень важно. Если обработка выполняется в несинхронизированном режиме, вероятно, что задание по созданию данных в восходящем направлении будет производить данные быстрее, чем другое задание по нисходящему потоку может их обработать. Когда это происходит, обработка должна блокировать, буферизовать или отбрасывать данные. Удаление данных, скорее всего, не вариант; блокировка может привести к остановке всего графа обработки. Журнал действует как очень, очень большой буфер, который позволяет перезапустить процесс или выйти из строя, не замедляя другие части графа обработки. Эта изоляция особенно важна при распространении этого потока данных на более крупную организацию, где обработка выполняется заданиями, выполняемыми множеством разных команд. У нас не может быть одной неисправной работы, вызывающей противодавление, которое останавливает весь процесс обработки.

концептуально кафка это **circular buffer** or **ring buffer** (те не бесконечный)

- Такой буфер также называют круговым или кольцевым. Однако вследствие своего нахождения на диске он может быть довольно большим
- журналирование — вариант буферизации с большим буфером фиксированного размера (ограниченным доступной емкостью диска)

способен служить буфером для сообщений за **11 часов**, после чего начнет перезаписывать старые сообщения.

- Сделаем приблизительный подсчет. На момент написания этой книги емкость типичного большого жесткого диска составляла 6 Тбайт, а скорость последовательной записи — 150 Мбайт/с.
- Если записывать сообщения с максимально возможной скоростью, то диск заполнится примерно через 11 часов.
- Таким образом, он способен служить буфером для сообщений за 11 часов, после чего начнет перезаписывать старые сообщения. При использовании нескольких дисков и машин это соотношение не изменится

На практике редко прибегают к полной скорости записи диска, так что журнал обычно способен хранить буфер за **несколько дней** или даже недель

- Независимо от того, как долго хранятся сообщения, скорость записи журнала остается более или менее постоянной, поскольку все сообщения в любом случае записываются на диск

кафка это не просто log, это **replayable log**

- Replayable logs decouple services from one another, much like a messaging system does, but they also provide a central point of storage that is fault-tolerant and scalable—a shared source of truth that any application can fall back to

с логами проще восстанавливать данные после сбоев

- Еще одна очень приятная особенность журнала, предназначенного только для добавления, заключается в том, что он позволяет гораздо проще восстанавливать данные после ошибок. Если вы развертываете некорректный код, который записывает неверные данные в базу данных, или если человек вводит некорректные данные, вы можете просмотреть журнал, чтобы увидеть точную историю того, что произошло, и отменить это. Такое восстановление намного сложнее, если вы перезаписали старые данные новыми или даже удалили данные неправильно

Традиционный подход к построению БД и схемы основан на ошибочном представлении о том, что данные должны записываться и выдаваться по запросу в одной и той же форме.

- Споры о нормализации и денормализации становятся во многом неактуальными, если есть возможность переводить данные из оптимизированного для записи журнала событий в оптимизированное для чтения состояние приложения.
- Это связано с тем, что вполне разумно денормализовать данные в оптимизированных для чтения представлениях, поскольку процесс преобразования предоставляет механизм, позволяющий обеспечивать его соответствие представлению с журналом событий.

потоковая обработка — это, прежде всего, технология управления данными (*data management technique*)

- журналы событий являются долговечными и многопользовательскими
- потоковые процессоры обычно объединяются в ациклические конвейеры, где каждый поток представляет собой выходные данные конкретной задачи и получен на основе четко определенного набора входных потоков

ТЕРМИНОЛОГИЯ

22 декабря 2020 г. 21:05

используемая терминология

- events как facts - для переноски состояния в event-carried state transfer
- events как triggers - для мгновенного уведомления других микросервисов чтобы они что-то начали сразу делать
- command - это синхронный вызов с побочным эффектом, к конкретному адресату
 - можно даже назвать его асинхронным но прямым односторонним вызовом доходящим сразу до адресата
 - вызовы даже могут быть http long pooling
 - Команды - это действия - запросы на выполнение какой-либо операции другой службой, что-то, что изменит состояние системы. Команды выполняются синхронно и обычно указывают на завершение, хотя могут также включать результат.
 - команда - это запрос на то, чтобы что-то произошло в будущем. Команда имеет явное ожидание, что что-то (изменение состояния или побочный эффект) произойдет в будущем. События происходят без таких ожиданий в будущем. Это просто заявление о том, что что-то произошло.
- query - это синхронный вызов без побочного эффекта, к конкретному адресату
 - те запросы направляются непосредственно к источнику данных.
 - Запросы - это просьбы найти что-то. В отличие от событий или команд, запросы не имеют побочных эффектов; они оставляют состояние системы без изменений.
- event - это асинхронный вызов, который может быть fact+trigger, без адресата (хотя главный адресат известено, но могут быть еще куча потом)
 - События - это факт, и уведомление. Они представляют собой нечто, что случилось в реальном мире, но не включают в себя ожиданий каких-либо дальнейших действий. Они движутся только в одном направлении и не ожидают ответа (иногда называемого «выстрелил и забыл»), но один может быть «синтезирован» из последующего события.
 - с точки зрения сервисов, события приводят к меньшей связи, чем команды и запросы. Слабая связь является желательным свойством, когда взаимодействия пересекают границы развертывания, поскольку службы с меньшим количеством зависимостей легче изменить.

fact event -

- we noticed that events, in fact, have two separate roles: one for notification (a call for action), trigger event
- and the other a mechanism for state transfer (pushing data wherever it is needed)

? отталкиваясь от терминологии command, query, event то получается что CQRS это только про HTTP от клиентского фронта

разное

event processing network (EPN)

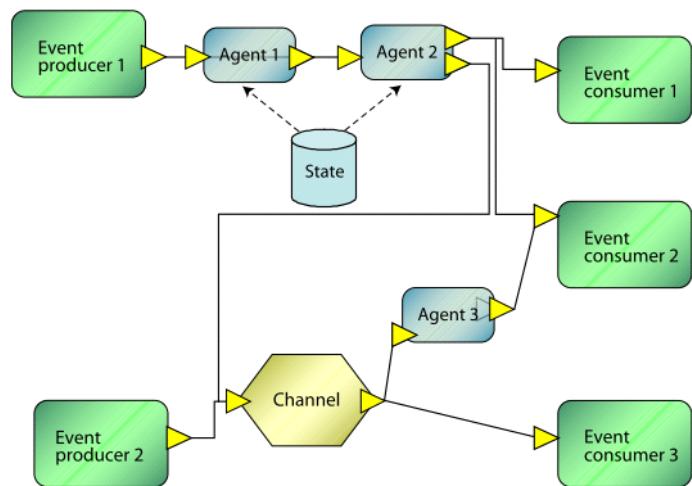


Figure 2.7 This is an example of a particular event processing network showing the event processing components. The event producers appear at the extreme left of the diagram, and the event consumers at the far right.

_кафка это лог (НА надежность + параллелизм)

12 декабря 2020 г. 12:19

```
cd C:\Users\voval\confluent-dev\labs  
docker-compose up -d
```

сама кафка называет себя как **Distributed Streaming Platform** (streams - означает log)
те кафка это не просто лог а **распределенный лог**

Apache Kafka is the foundation of what we call a Distributed Streaming Platform. Kafka is used for building real-time data pipelines and streaming apps. It is horizontally scalable, fault-tolerant, wicked fast, and runs in production in thousands of companies.

A streaming platform has three key capabilities:

- Publish and subscribe to streams of records, similar to a message queue or enterprise messaging system.
- Store streams of records in a fault-tolerant durable way.
- Process streams of records as they occur, in real-time.

Kafka is generally used for two broad classes of applications:

- Building real-time streaming data pipelines that reliably get data between systems or applications
- Building real-time streaming applications that transform or react to the streams of data

распределенность нужна для восстановления из сбоя и для скорости (см ниже НА и параллелизм)

- Распределенные журналы фиксаций служат именно для восстановления после сбоя

партиционированный лог (тк топик состоит из партиций)

KAFKA = PARTITIONED
TOPIC = LOG

(1) **HA (high availability)** /надежность достигается за счет репликации (копий партиции)
reliability

- для скорости не годится так как клиенты не читают с копий а только с лидер-партиции
- Kafka обеспечивает надежность за счет репликации. Это означает, что сообщения записываются на настраиваемое количество машин, так что, если одна или несколько из этих машин выйдут из строя, сообщения не будут потеряны. Если вы установите коэффициент репликации равный трем, две машины могут быть потеряны без потери данных.
- **Durability** В едином журнале будут реплицироваться все события в кластере. Без этого распределения событий единственный журнал был бы уязвим для потери данных.

(2) **параллелизм/scalability/скорость** достигается за счет partitioning (деление топика на партиции)
performance

Паттерн «один потребитель из расчета на одну секцию» позволяет добиться максимальной пропускной способности

- Kafka uses replication to achieve HA. Each partition of a topic is available in the cluster on say 3 distinct brokers. In case of failure of one broker, still two additional copies of the data are available.
- Основное преимущество этого с архитектурной точки зрения заключается в том, что он снимает проблему масштабируемости со стола. С Kafka практически невозможно преодолеть стену масштабируемости в контексте бизнес-систем. Это может быть очень полезным, особенно когда экосистемы растут, позволяя разработчикам выбирать шаблоны, которые немного более свободны с пропускной способностью и перемещением данных.
- Масштабируемость открывает и другие возможности. Отдельные кластеры могут расти до масштабов компаний без риска перегрузки инфраструктуры рабочими нагрузками
- Stateful stream processing systems like Kafka Streams avoid remote transactions or cross-process coordination. They do this by partitioning the problem over a set of threads or processes using a chosen business key. This provides the key (no pun intended) to scaling these systems horizontally

так как все затевалось ради автоматического

- параллелизма
- load balancing (механизм чтобы партиции равномерно распределялись между консьюмерами)
- fault tolerance (механизм чтобы сообщения в партициях не скапливались)

то лучше оставить этот механизм **автоматическим**, и НЕ делать жесткие привязки сообщений к партициям и партиций к консьюмерам

а самому подумать над state management

- state management - даже если новый инстанс получит своих клиентов назад, то надо как то подумать о передаче старого-уже-накопленного-результатата-счетчика от умершего инстанса к новорожденному
- а если новый инстанс получит новых клиентов то всеравно state-management ему нужно подумать как получить предыдущий итоговый счетчик по этим клиентам

- При секционировании топика отправляемые в него данные, по существу, разбиваются на параллельные потоки. Именно в этом и заключается секрет потрясающей пропускной способности Kafka. Я уже упоминал, что топик — это распределенный журнал; каждая секция сама по себе тоже является журналом и следует тем же правилам. Kafka добавляет каждое из входящих сообщений в конец журнала, и все эти сообщения оказываются строго упорядоченными по времени. Каждому сообщению присваивается смещение. Упорядоченность гарантируется только внутри секции, но не между различными секциями.
- Секционирование служит и другой цели, помимо повышения пропускной способности. Благодаря ему можно распределять сообщения топиков по нескольким машинам, так что **вместимость конкретного топика не будет ограничиваться дисковым пространством, доступным на одном сервере.**
- Scalability - аспределение единого журнала по кластеру машин позволяет нам работать с потоками событий, размер которых превышает емкость любой отдельной машины. Это важно, потому что любой заданный поток событий (например, данные телеметрии такси из нашего предыдущего примера) может быть очень большим. Распределение по кластеру также упрощает кластеризацию каждого приложения, читающего единый журнал

(3) solve bigger problems - когда все данные не помещаются на одну машину

- те их по-любому придется разбить на части

лог - это просто файл (segment)

- The log-structured approach is itself a simple idea: a collection of messages, appended sequentially to a file.
- Кластер Kafka - это, по сути, набор файлов, заполненных сообщениями, охватывающими множество разных машин. Большая часть кода Kafka включает связывание этих различных индивидуальных журналов вместе, надежную маршрутизацию сообщений от производителей к потребителям, репликацию для обеспечения отказоустойчивости и корректную обработку сбоев.
- Как и многие другие хорошие результаты в информатике, эта масштабируемость во многом объясняется простотой. Базовая абстракция представляет собой секционированный журнал - по сути, набор файлов только для добавления, распределенных по нескольким машинам, - который поощряет шаблоны последовательного доступа, которые естественным образом соответствуют структуре базового оборудования.
- Итак, журнал ничем не отличается от файла или таблицы. Файл - это массив байтов, таблица - это массив записей, а журнал - это просто своего рода таблица или файл, в котором записи сортируются по времени

лог - это упорядоченная по времени и только для добавления последовательность

- Журнал - это, пожалуй, самая простая абстракция хранилища. Это полностью упорядоченная последовательность записей, **упорядоченная по времени, только для добавления**
- Записи добавляются в конец журнала, и чтение выполняется слева направо. Каждой записи присваивается уникальный порядковый номер записи в журнале.
- Порядок записей определяет понятие «время», поскольку записи слева определены как более старые, чем записи справа. Номер записи в журнале можно рассматривать как «метку времени» записи. Поначалу описание этого порядка как понятия времени кажется немного странным, но оно имеет удобное свойство, заключающееся в том, что оно не связано с какими-либо конкретными физическими часами. Это свойство окажется важным, когда мы перейдем к распределенным системам.
- Ответ заключается в том, что журналы служат для определенной цели: **они фиксируют, что и когда произошло**
- В контексте Kafka (или любой другой распределенной системы) журналом называется «предназначенная только для дописывания в ее конец, полностью упорядоченная по времени последовательность записей»
Приложение добавляет записи в конец журнала по мере их поступления. Записи упорядочиваются по времени, хотя метки даты/времени в них может и не быть, просто слева располагаются записи, поступившие первыми, а справа — последними.

топик можно рассматривать как маркированный лог

- Топики в Kafka представляют собой журналы, разбитые в соответствии с названием топика.
- Топики можно рассматривать практически как маркированные журналы

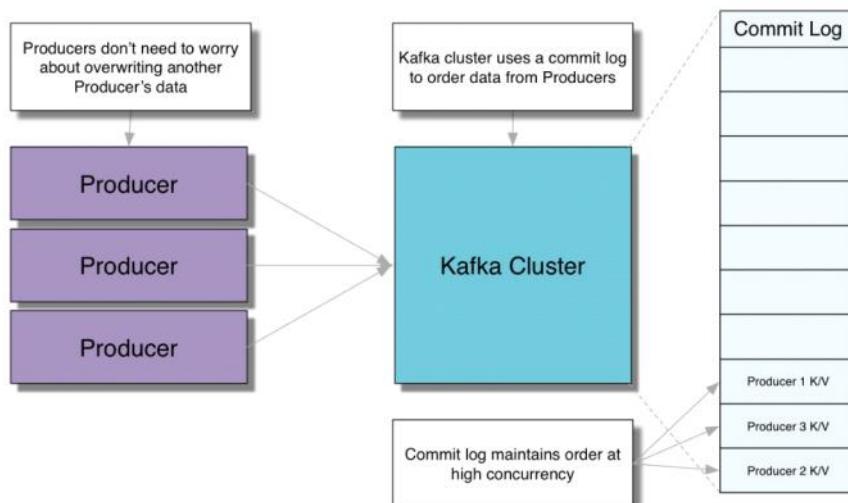
O(1) для последовательной записи или чтения с концов

Но журнал имеет значение O (1) при чтении или записи сообщений в раздел, поэтому гораздо меньше имеет значение, находятся ли данные на диске или кэшированы в памяти.

commit log - понятие пришедшее из мира БД

- Inbound producer requests are placed in a request queue when received by the broker. This allows the broker to handle "simultaneous" messages without loss of data and to avoid throttling inbound traffic if possible. The broker moves the data from the request queue into the commit log which resides in the page cache.
- Одна из замечательных особенностей этого подхода заключается в том, что временные метки, которые индексируют журнал, теперь действуют как часы для состояния реплик - вы можете описать каждую реплику одним числом - меткой времени для максимальной записи журнала, которую она обработала. Эта временная метка в сочетании с журналом однозначно фиксирует все состояние реплики
- Например, концепция журнала дает логические часы для каждого изменения, по которым можно измерить всех подписчиков. Это значительно упрощает рассуждения о состоянии различных абонентских систем по отношению друг к другу, поскольку у каждой есть «момент времени», который они прочитали.

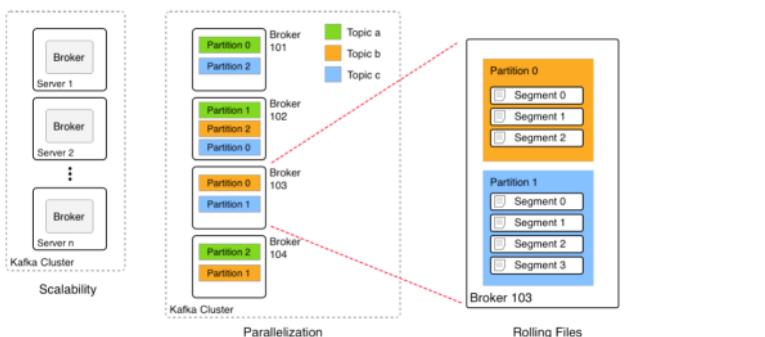
The Commit Log for High Concurrency



распределенный лог

- Each partition within a topic is a subset of the data held by that topic. Therefore, each partition will have its own commit log, each with its own set of messages and offset numbers.
- Each commit log is subdivided into physical files on disk called segments. This is needed due to the fact that the amount of data that can be written to a partition is unbounded.
- Две проблемы, которые решает журнал, - упорядочивание изменений и распределение данных - еще более важны в распределенных системах данных.

Partitions Are Stored as Separate Logs



- журнал унифицирован или распределен? Собственно, и то, и другое! Распределенный и унифицированный относятся к различным свойствам журнала. Журнал унифицирован, потому что реализация единого единого журнала лежит в основе бизнеса, как объяснялось ранее в разделе 2.1.1. Единый журнал распространяется, потому что он находится в кластере отдельных машин.

иногда лог сравнивают с [системой контроля версий](#) (которая тоже основана на логе), потому что можно откатиться и сделать [replay](#) для исследования ошибки

Если мы сохраняем события в журнале, он начинает вести себя как система контроля версий для наших данных. Например, если вы развернете неисправную программу, система может быть повреждена, но ее всегда можно будет восстановить. Последовательность событий обеспечивает точку аудита, чтобы вы могли точно выяснить, что произошло. Кроме того, когда ошибка будет исправлена, вы можете перемотать назад свою службу и снова начать обработку.

[лог - более фундаментальная структура данных чем таблица, тк из лога можно получить любые виды таблиц](#)

- . В некотором смысле журнал является более фундаментальной структурой данных: помимо создания исходной таблицы вы также можете преобразовать ее для создания всех видов производных таблиц. (И да, таблица может означать хранилище данных с ключами для нереляционных людей.)

[лог это очень простая и единообразная структура данных, которую легко обрабатывать \(те неспроста укали такую простой API\)](#)

- Эти данные должны быть смоделированы единообразно, чтобы их было легко читать и обрабатывать.
- Журналы — простая абстракция данных, имеющая очень большое значение. При упорядоченных по времени записях разрешение конфликтов и выбор нужного обновления для различных машин сильно упрощаются: просто выбирается последняя запись.

[те ответственность за нужный формат данных теперь мудро лежит на том кому эти данные нужны](#)

- классическая проблема группы хранилища данных заключается в том, что они отвечают за сбор и очистку всех данных, созданных любой другой командой в организации. [Стимулы не согласованы](#): производители данных часто не очень осведомлены об использовании данных в хранилище и в конечном итоге создают данные, которые трудно извлечь или которые требуют тяжелого, трудно масштабируемого преобразования для получения пригодной для использования формы. Конечно, центральной группе никогда не удается полностью масштабироваться, чтобы соответствовать темпам остальной части организации, поэтому охват данных всегда неравномерный, поток данных хрупкий, а изменения происходят медленно.
- Лучший подход - иметь центральный конвейер, [журнал, с четко определенным API для добавления данных](#). Ответственность за интеграцию с этим конвейером и предоставление [чистого, хорошо структурированного потока данных лежит на производителе этого потока данных](#). Это означает, что в рамках проектирования и реализации системы они должны учитывать проблему вывода данных и их преобразования в хорошо структурированную форму для доставки в центральный конвейер. Добавление новых систем хранения не имеет значения для команды хранилищ данных, поскольку у них есть центральная точка интеграции. Команда хранилища данных решает только более простую задачу загрузки структурированных каналов данных из центрального журнала и выполнения преобразования, специфичного для их системы.

[Объединение и разделение — две стороны одной медали: надежной, масштабируемой и удобной в сопровождении системы, построенной из различных компонентов.](#)

принцип разделения соответствует традициям Unix: малые инструменты, хорошо выполняющие одну конкретную операцию [22] и обменивающиеся данными через унифицированный API (конвейеры) низкого уровня и которые можно объединять, используя языки более высокого уровня

Объединенные базы данных: [унифицированное чтение](#).

- Можно обеспечить единый интерфейс запросов для широкого круга базовых механизмов хранения и методов обработки — подход, известный как объединенная база данных или полихранилище [18, 19]. Например, этому паттерну соответствует адаптер внешних данных в PostgreSQL [20]. Приложения, которые нуждаются в специализированной модели данных или интерфейсе запросов, по-прежнему могут напрямую обращаться к основным хранилищам, тогда как пользователям, желающим объединить данные из разных мест,

легко сделать это через объединенный интерфейс. Интерфейс объединенного запроса соответствует реляционной традиции единой интегрированной системы с языком запросов высокого уровня и элегантной семантикой, но имеет сложную реализацию.

- Объединенные запросы на чтение требуют сопоставления данных из двух моделей.

Разделенные базы данных: унифицированная запись.

- Идея объединения баз касается запросов на чтение в разных системах, она не предлагает хороших решений для синхронизации записи в этих системах. Как уже отмечалось, создание согласованного индекса в единой БД является встроенной функцией. Но при объединении нескольких систем хранения информации также необходимо гарантировать, что все измененные данные будут попадать в нужные места даже в случае сбоев. Упрощение надежного подключения систем хранения (например, путем сбора данных об изменениях и создания журналов событий) подобно разделению функций обслуживания индекса в базе таким образом, чтобы можно было синхронизировать записи, сделанные по разным технологиям [7, 21]

Каждый логический источник данных можно смоделировать как собственный журнал.

Источником данных может быть приложение, которое регистрирует события (например, клики или просмотры страниц), или таблица базы данных, которая принимает изменения. Каждая подписавшаяся система читает из этого журнала как можно быстрее, применяет каждую новую запись к своему собственному хранилищу и продвигает свою позицию в журнале

подписчик потребляет в темпе, который он контролирует.

Журнал также действует как буфер, который делает производство данных асинхронным по сравнению с потреблением данных. Это важно по многим причинам, но особенно когда есть несколько подписчиков, которые могут потреблять с разной скоростью. Это означает, что система подписки может выйти из строя или выйти из строя для обслуживания и наверстать упущенное, когда она вернется: подписчик потребляет в темпе, который он контролирует.

самая большая проблема, которую решает лог - это просто сделать данные доступными в потоках данных нескольким подписчикам в реальном времени (just making data available in real-time multi-subscriber data feeds.)

Оказывается, журнал решает некоторые из наиболее важных технических проблем в потоковой обработке, которые я опишу, но самая большая проблема, которую он решает, - это просто сделать данные доступными в потоках данных нескольких подписчиков в реальном времени.

Журнал — простейший способ обеспечить рассылку сообщений

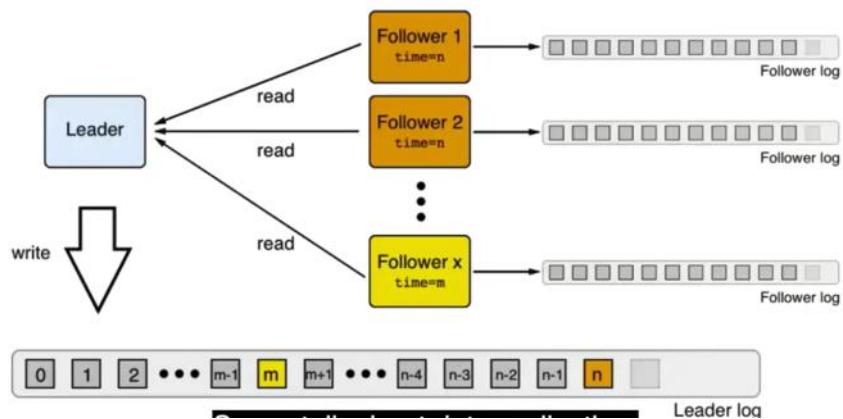
- несколько пользователей могут читать журнал независимо друг от друга, при этом прочитанные сообщения не удаляются из него.
- Для обеспечения равномерного распределения нагрузки в группе потребителей брокер, вместо того чтобы направлять отдельные сообщения клиентам-потребителям, может назначать целые разделы узлам в потребительской группе.
- Затем каждый клиент получает все сообщения из того же раздела, который был ему назначен. Обычно, когда потребителю назначается раздел журнала, он читает сообщения последовательно, простым однопоточным способом. Этот простейший вариант распределения нагрузки имеет следующие недостатки:
 - количество узлов-потребителей, совместно обрабатывающих одну тему, не может быть больше количества разделов журнала в этой теме
 - если какое-то сообщение медленно обрабатывается, то задерживает обработку следующих сообщений этого раздела-партиции

single leader replication (follower получается как еще одна из форм потребителя)

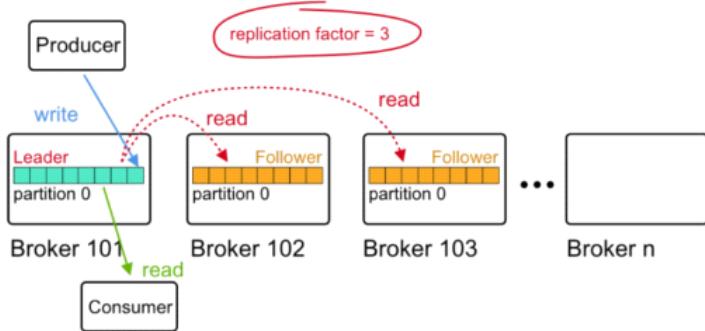
- каждый follower увеличивает availability
- репликация происходит на уровне партиций (те одна партиция главная в топике и остальные реплики)

One possible way of doing a log replication is to define a (single) leader and a number of followers (Note: this is what Kafka does). The leader writes and keeps the principal copy of the log. The followers read from the leader's log (managed by the leader) and update their own local log accordingly, as shown in the image.

Log Replication



Replication of Partitions



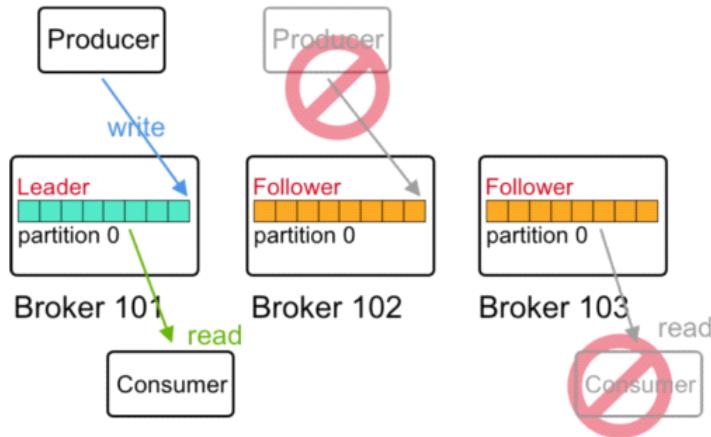
- Kafka maintains replicas of each partition on other Brokers in the cluster
 - Number of replicas is configurable

- One Broker is in the role of **leader** for a particular partition
 - All writes and reads go to and from the leader
 - Other Brokers are in the role of **follower** for that partition

At any given point of time, all the replicas are byte wise identical (except for where one hasn't caught up). To facilitate that, the ordering of the messages must be identical on all replicas.

For any given replicated partition, one replica is a leader and the rest are followers. All I/O (produce and consumer requests) go to the leader. In the case of write requests, this allows just one replica (the leader) to determine the ordering of messages. Once the messages are written to the local log of the leader, all the replicas can get the data.

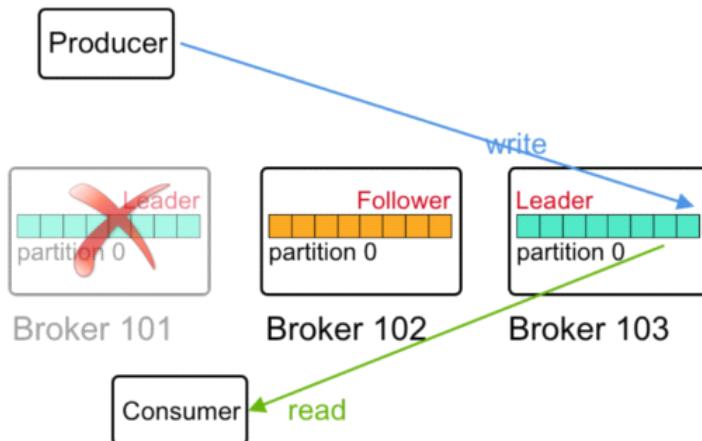
Important: Clients Do Not Access Followers



- It is important to understand that Producers only write to the leader
- Likewise, Consumers *only* read from the leader
 - They do not read from the replicas
 - Replicas only exist to provide reliability in case of Broker failure
- The followers just copy the data from the commit log of the leader as a pull request; they do not interact with any external clients.

i KIP-392 may change the behavior of clients (consumers) being able to access followers in a future version: <https://cwiki.apache.org/confluence/display/KAFKA/KIP-392%3A+Allow+consumers+to+fetch+from+closest+replica>

Leader Failover



- If a leader fails, the Kafka cluster will elect a new leader from among the followers
- The clients will automatically switch over to the new leader

репликация = сколько копий у партиции будет (напоминание и разделение)

топика на партиции отношение не имеет)

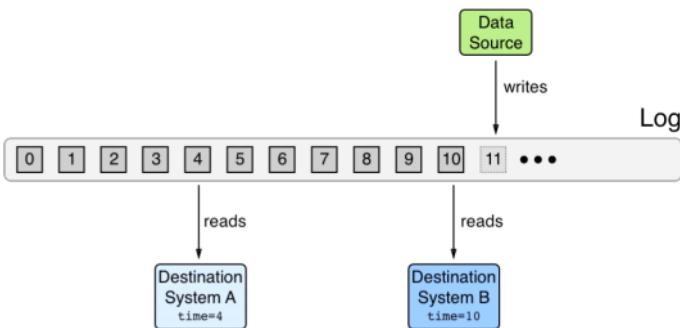
- если у нас в топике 3 партиции и replication factor = 2 то в итоге будет 6 логов
- каждая копия/реплика должны быть на разной ноде

Partitioning data is good for performance but not for reliability. The more parts there are in any system, the higher the probability that one of them will fail. Kafka addresses this problem by included a built -in replication solution to maintain multiple copies of each partition



write once -> read many, тип структуры данных (тк лог-немутабельный) подходящий когда много потребителей

- Multiple destination systems can independently consume from the log, each at its own speed. In the sample we have two consumers that read from different positions at the same time. This is totally fine and an expected behavior.



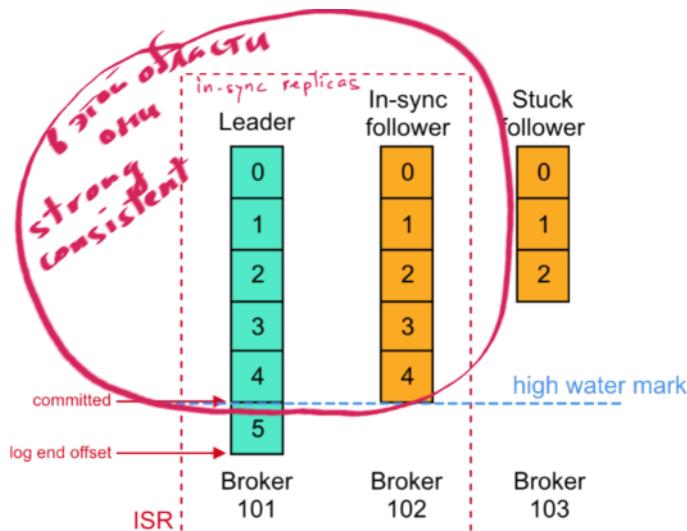
strongly consistent system - на самом деле в кафке **строгая консистентность**

- те реплики в кафке которые полностью идентичны по данным (в данный момент времени) называются **in-sync replicas**

- а не eventualy consistency
- The followers are up to date with the leader if their last offset (or time) in their private log corresponds to the last offset of the leader's log. A follower is behind,

or out of sync if its offset is behind the last offset of the leader. By default, for Kafka this is true, when a follower is more than 10s behind the end of the leader's log. Adjust the parameter `replica.lag.time.max.ms` to change this default.

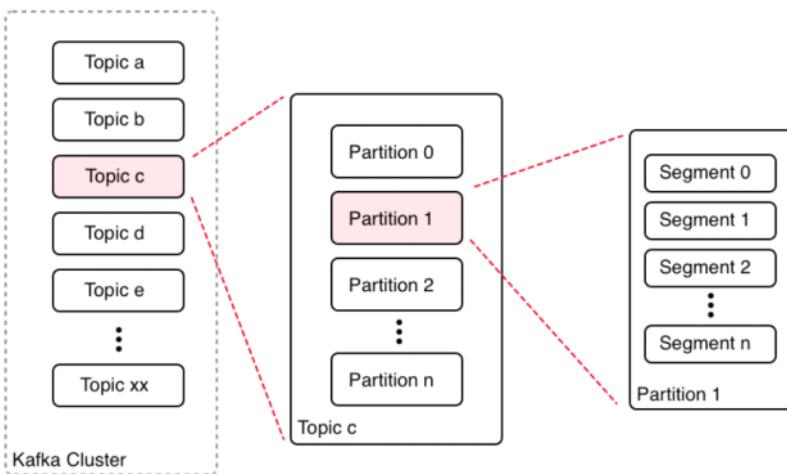
- The followers whose log is up to date with the leader are called **in-sync replicas**. Any of those followers can take over the role of a leader if the current leader fails.



topics -> partitions -> segments

- scalability также достигается за счет partitioning (но только когда партиции размазаны по брокерам а НЕ когда все они на одном сервере)
- topic в кафке распределенный (те размазан по нодам)
- партиция - это просто часть от общих данных (обычно в БД это называется sharding)
- на самом деле под логическое определение лога наиболее всего подходит **партиция**

To parallelize work and thus increase the throughput Kafka can split a single topic into many partitions. The messages of the topic will then be split between the partitions. The default algorithm used to decide to which topic partition a message goes uses the hash code of the message key. A partition is handled in its entirety by a single Kafka broker. A partition can be viewed as a "log"



разделение на партиции является основной фичей чем кафка отличается от простого pub-sub (также гарантия упорядоченности внутри партиции)

- Of course, separating publishers from subscribers is nothing new. But if you want to keep a commit log that acts as a multi-subscriber real-time journal of everything happening on a consumer-scale website, scalability will be a primary challenge.
- Отсутствие глобального порядка между разделами является ограничением, но мы не считаем его серьезным. Действительно, взаимодействие с журналом обычно происходит от сотен или тысяч отдельных процессов, поэтому бессмысленно говорить об общем порядке их поведения. Вместо этого гарантии, которые мы предоставляем, заключаются в том, что каждый раздел сохраняет порядок, а Kafka гарантирует, что добавление к определенному разделу от одного отправителя будет доставлено в том порядке, в котором они были отправлены.

имплементации (те нарезание на сегменты) нужно чтобы файл не оказался бесконечным

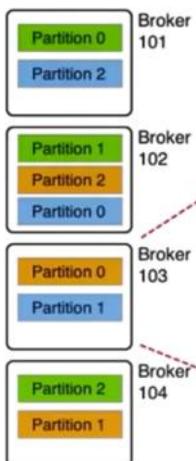
- **то чтобы обеспечить retention policy** (удаление старых сегментов или compaction)
- в один момент времени может быть только один активный сегмент

The broker stores the messages as they come in in a physical file. Since the data can potentially be endless the broker is using a "rolling-file" strategy. It creates/allocates a new file and fills it with messages. When the segment is full (or the given max. time per segment expires), the next one is allocated by the broker.

Kafka opens/allocates a file on disk and then fills that file sequentially and in append only mode until it is full, where "full" depends on the defined max size for the file, (or the defined max time per segment is expired).

кафка стремится сделать так чтобы партиции раскидывались по разным брокерам

- это нужно чтобы достичь надежности



Isolation is a prerequisite for resilience and elasticity

- and requires asynchronous communication boundaries between services to decouple them in:
- Изоляция - самая важная черта. Это основа для многих высокочувствительных преимуществ микросервисов. Но это также черта, которая оказывает наибольшее влияние на ваш дизайн и архитектуру
- Изоляция - необходимое условие автономии. Только когда службы изолированы, они могут быть полностью автономными и принимать решения независимо, действовать независимо, а также сотрудничать и координировать свои действия с другими для решения проблем.
- Другой аспект автономии заключается в том, что если служба может давать обещания только о своем собственном поведении, тогда вся информация, необходимая для разрешения конфликта или устранения сценариев сбоя, доступна внутри самой службы, что устраняет необходимость в общении и координации. .

Time

Allowing concurrency

Space

Allowing distribution and mobility—the ability to move services around

примеры

- **Failure isolation**—to contain and manage failure without having it cascade throughout the services participating in the workflow—is a pattern sometimes referred to as Bulkheading.
- **Isolation between services** makes it natural to adopt Continuous Delivery. This allows you to safely deploy applications and roll out and revert changes incrementally—service by service.
- **Isolation also makes it easier to scale each service**, as well as allowing them to be monitored, debugged and tested independently—something that is very hard if the services are all tangled up in the big bulky mess of a monolith.

пределный случай шардинга/партиционирования

Таким образом, первым важным моментом при реализации реактивного приложения является определение минимальной единицы обработки, которая может работать **независимо**

- В реактивном приложении каждая часть - каждая независимая единица обработки - **реагирует на полученную информацию**. Поэтому очень важно учитывать, какие потоки

информации, где и насколько велик каждый из этих потоков.

пример входных данных от клиентов (interactive google maps)

- В примере приложения вы можете определить, что каждый пользователь отправляет одно обновление позиции каждые 5 секунд, пока приложение работает на их мобильном устройстве - телефоне, планшете или часах. Вы можете убедиться в этом, написав клиентское приложение самостоятельно или установив это ограничение в API, который приложение предлагает автору клиентских приложений. Каждое обновление позиции будет составлять примерно 100 байтов, плюс-минус (по 10 байт для метки времени, широты и долготы; 40 байт для служебных данных протокола низкого уровня, например TCP / IPv4.); плюс дополнительное место для шифрования, аутентификации и данных целостности). Если учесть некоторые накладные расходы на предотвращение перегрузки и планирование сообщений между несколькими клиентами, вы предположите, что поток обновления позиции каждого клиента стоит в среднем примерно 50 байт в секунду

пример выходных данных доставляемых клиентам

- Размещение фрагмента карты на одном сетевом узле означает возможность обслуживать 500 000 клиентов в пределах этой области карты. Поэтому плитки должны быть достаточно маленькими, чтобы этот предел никогда не нарушился. Если все фрагменты карты имеют одинаковый размер, т. Е. Если на всей карте используется один и тот же уровень разделения, то некоторые фрагменты будут посещаться гораздо чаще, чем другие. Густонаселенные районы, такие как Манхэттен, Сан-Франциско и Токио, будут близки к пределу, тогда как по большей части плиток, покрывающих Тихий океан, никто не будет двигаться по ним. Эту асимметрию можно учесть, разместив несколько листов карты с низкой скоростью на одном узле обработки, сохранив листы с высокой скоростью на их собственных узлах
- Напомним, что для интерфейсных узлов критически важно иметь возможность выполнять свою работу независимо друг от друга, чтобы иметь возможность регулировать пропускную способность системы путем добавления или удаления узлов; вы увидите еще одну причину этого, когда мы обсудим, как реагировать на сбои в системе. Но как достичь консенсуса относительно того, какой фрагмент карты размещен на каком внутреннем сетевом узле? Ответ заключается в том, что вы делаете процесс маршрутизации простым и детерминированным, заставляя службу распределения листов карты распределять структуру данных, которая описывает размещение всех листов. Эта структура данных может быть оптимизирована и ската с помощью иерархической структуры разбиения карты.
- Когда вы визуализируете карту, которая будет показывать движение всех анонимных пользователей в пределах ее области, что вы ожидаете увидеть при уменьшении масштаба? Вы, конечно, не можете следить за каждым человеком и отслеживать его курс, если их больше, чем горстка в регионе, на который вы смотрите. А когда вы уменьшаете масштаб, чтобы увидеть всю Европу, лучшее, на что вы можете надеяться, - это совокупная информация о плотности населения или средней скорости движения - вы не сможете различить отдельные позиции.

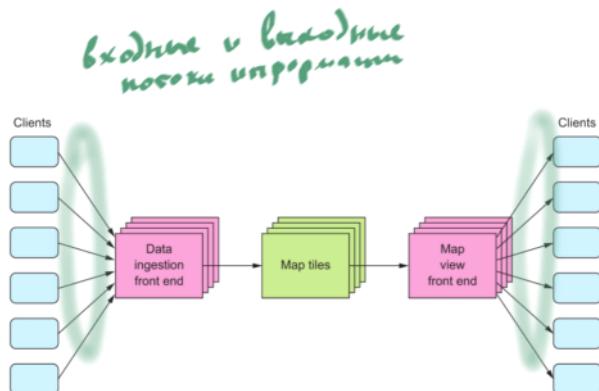


Figure B.2 Data are flowing from the clients that submit their position updates through a front end that handles the client connections into the map tiles and on to those clients that are watching the map.

(1) один человек = один микросервис

Grouping data and behavior according to transaction boundaries

- Чтобы формализовать то, что мы только что обсудили, трюк состоит в том, чтобы нарезать поведение и сопутствующий набор данных таким образом, чтобы каждый фрагмент предлагал желаемые функции изолированно, и не требовались транзакции, охватывающие несколько фрагментов. Этот метод применяется и подробно обсуждается в литературе по предметно-ориентированному проектированию (DDD)

вся почта будет буквально храниться в этом экземпляре. Это означает, что весь доступ к содержимому электронной почты человека будет осуществляться через этот выделенный экземпляр.

- The example problem of storing a person's email folders in a distributed fashion can be solved by applying the strategy outlined in the previous section. If you want to ensure that emails can be moved without leading to inconsistent counts, then each person's complete email dataset must be managed by one entity. The example application decomposition would have a module for this purpose and would instantiate it once for every person using the system. This does not mean all mail would be literally stored within that instance. It only means all access to a person's email content would be via this dedicated instance

Это нормально, потому что человек на много порядков медленнее, чем компьютер, когда дело касается обработки электронной почты, поэтому вы не столкнетесь с проблемами производительности, ограничив масштабируемость в этом направлении

In effect, this acts like a locking mechanism that serializes access, with the obvious restriction that an individual person's email cannot be scaled out to multiple managers in order to support higher transaction rates. This is fine, because a human is many orders of magnitude slower than a computer when it comes to processing email, so you will not run into performance problems by limiting scalability in this direction. What is more important is that this enables you to distribute the management of all users' mailboxes across any number of machines, because each instance is independent of all others. The consequence is that it is not possible to move emails between different people's accounts while maintaining the overall email count, but that is not a supported feature anyway.

[2] один внешний ресурс == один микросервис

- вы должны определить тот компонент, в чью ответственность входит каждый ресурс, и поместить его туда. Ресурс становится частью ответственности этого компонента
- конкретный Ресурс и его жизненный цикл - это обязанности, которые должны принадлежать одному компоненту
- паттерн [Resource Encapsulation](#) [onenote..\BOOK%20Hanafee%20Brian%20Reactive%20Design%20Patterns.one#%20Resource%20Encapsulation%20pattern§ion-id={5E93C244-52DD-41ED-AD98-A8EF049499EF}&page-id={E27FA7F6-8064-4010-B6D0-6E3208724684}&end&base-path=C:\Users\trans_Qsync\voya_from_onenote\tf_algonote_v1\SYSTEMDESIGN](#)
- The Resource Encapsulation pattern is used in two cases: **to represent an external resource and to manage a supervised resource**—both in terms of its lifecycle and its function, in accordance with the Simple Component pattern and the principle of delimited consistency.
- Вы инкапсулировали ресурсы, используемые вашей системой, в компоненты, которые управляют, представляют и напрямую реализуют свои функции. Это позволяет ограничить ответственность не только соображениями модульности кода (глава 6), но и вертикальной и горизонтальной масштабируемостью (главы 4 и 5) и принципиальной обработкой сбоев (глава 7). Цена всех этих преимуществ заключается в том, что вы вводите границу между ресурсом и остальной системой, которую можно пересечь только с помощью асинхронной передачи сообщений.

особенно этот паттерн применим, когда у ресурсов есть жизненный цикл которым необходимо управлять

- Шаблон применим везде, где ресурсы интегрированы в систему, в частности, когда у этих ресурсов есть жизненный цикл, которым необходимо управлять или представлять.

это дает также свободу переключаться на другого поставщика услуг с таким же интерфейсом

- вы не только инкапсулируете ответственность за преодоление превратностей доступности внешней службы, но также можете переключиться на совершенно другого поставщика услуг инфраструктуры, заменив это единственное внутреннее представление.

это дает возможность управлять квотами на ресурс

- Другой аспект такого распределения ответственности заключается в том, что это естественное и единственное место, где вы можете реализовать управление квотами на вызовы службы: если API инфраструктуры налагает ограничения на частоту выполнения запросов, вы будете отслеживать запросы, проходящие через этот путь доступа. Это позволит вам откладывать запросы, чтобы избежать временного превышения, которое может привести к штрафному ухудшению качества обслуживания

* про удаление

22 февраля 2021 г. 15:15

из лога нельзя удалить данные - означает что их нельзя удалить произвольно

- События автоматически удаляются из единого журнала, когда их возраст превышает установленный временной интервал, но их нельзя удалить произвольно
- Доступность только для добавления означает, что вашим приложениям намного проще анализировать свое взаимодействие с единым журналом. Если ваше приложение считало события до события номер 10 включительно, вы знаете, что вам никогда не придется возвращаться и снова смотреть на события с 1 по 10
- Конечно, возможность только добавления создает свои проблемы: если вы допустили ошибку при генерации событий, вы не можете просто войти в единый журнал и применить изменения для исправления этих событий, как вы могли бы это сделать в обычной реляционной базе данных или базе данных NoSQL

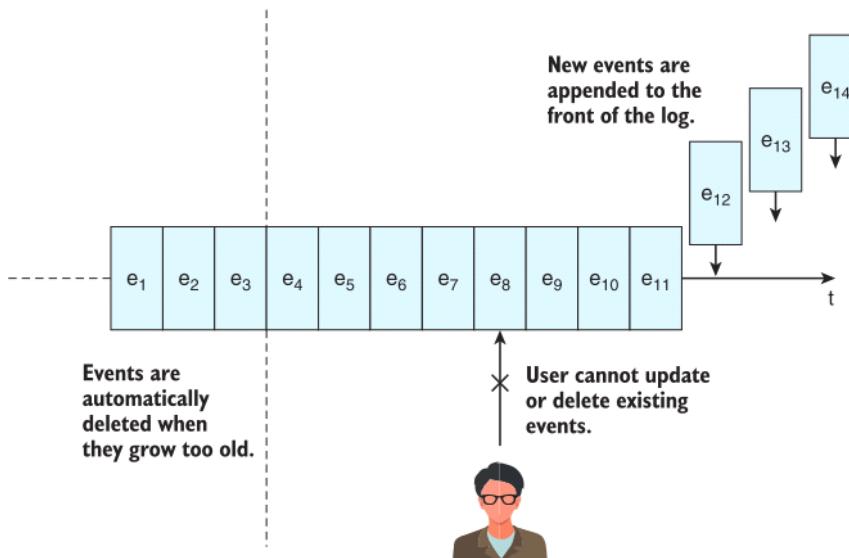


Figure 2.2 New events are appended to the front of the log, while older events are automatically deleted when they age beyond the time window supported by the unified log. Events already in the unified log cannot be updated or deleted in an ad hoc manner by users.

(1) хранить, если обнаружится ошибка в коде , через неделю

можно хранить данные вечно (на самом деле надо хранить ровно столько, сколько нужно для повторной обработки)

- Cheap consumers and the ability to retain large amounts of data make adding the second “reprocessing” job just a matter of firing up a second instance of your code but starting from a different position in the log.
- если вы хотите повторно обрабатывать данные за 30 дней, установите срок хранения в Kafka на 30 дней
- Kafka поддерживает репликацию и отказоустойчивость, работает на дешевом стандартном оборудовании и с радостью хранит много ТБ данных на каждой машине. Таким образом, сохранение больших объемов

данных - это совершенно естественный и экономичный способ, который не повредит производительности. LinkedIn хранит в сети более петабайта хранилища Kafka, и ряд приложений эффективно используют этот шаблон длительного хранения именно для этой цели.

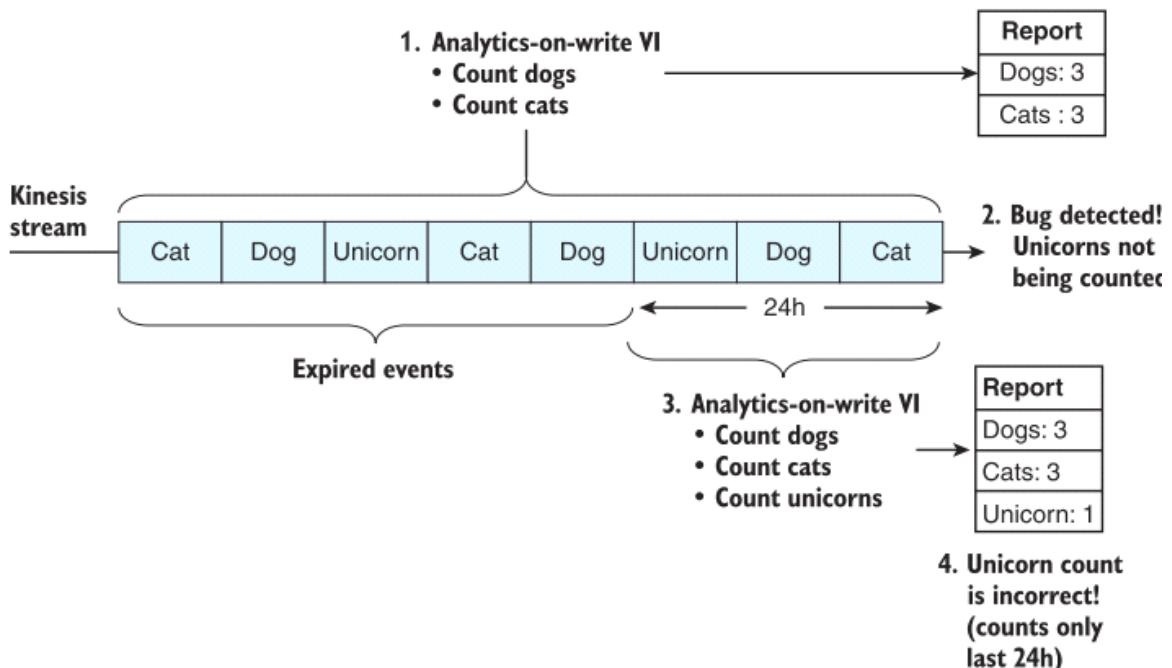


Figure 11.2 In the case of a bug in our analytics-on-write implementation, we have to fix the bug and redeploy the analytics against our event stream. At best, we can recover the last 24 hours of missing data, as depicted here. At worst, our output is corrupted, and we have to restart our analytics from scratch.

(2) хранить хотябы немножко, на случай если сервис упадет, чтобы второй спел поднялся

- Журнал действует как очень, очень большой буфер, который позволяет перезапустить процесс или выйти из строя, не замедляя другие части графа обработки
- Во-вторых, журнал обеспечивает буферизацию процессов. Это очень важно. Если обработка выполняется в несинхронизированном режиме, вероятно, что задание по созданию данных в восходящем направлении будет производить данные быстрее, чем другое задание по нисходящему потоку может их обработать. Когда это происходит, обработка должна блокировать, буферизовать или отбрасывать данные. Удаление данных, скорее всего, не вариант; блокировка может привести к остановке всего графа обработки. Журнал действует как очень, очень большой буфер, который позволяет перезапустить процесс или выйти из строя, не замедляя другие части графа обработки. Эта изоляция особенно важна при распространении этого потока данных на более крупную организацию, где обработка выполняется заданиями, выполняемыми множеством разных команд. У нас не может быть одной неисправной работы, вызывающей противодавление, которое останавливает весь процесс обработки.

Единственный побочный эффект обработки, кроме генерации потребителем выходных данных, — увеличение потребительского смещения.

- Но смещением управляет потребитель, поэтому при необходимости им легко манипулировать: например, можно запустить копию потребителя со вчерашними смещениями и записать выходные данные в другое место, чтобы заново обработать сообщения последнего дня. Такую операцию можно повторять многократно, изменяя код обработки.

(3) хранить столько, чтобы даже медленный консьюмер успел прочитать

если медленный потребитель не справляется со скоростью поступления сообщений и настолько отстает, что его потребительское смещение указывает на удаленный сегмент, то он пропустит некоторые сообщения.

- По сути, журнал представляет собой буфер ограниченного размера, отбрасывающий старые сообщения по мере заполнения.
- При отставании потребителя настолько сильно, что требуемые ему сообщения старше сохраненных на диске, он не сможет прочитать эти сообщения: брокер отбрасывает сообщения, хранящиеся слишком долго, если размер буфера не позволяет хранить их дольше.
- Можно следить за тем, насколько потребитель отстает от начала журнала, и выдать предупреждение, если отставание значительное
- Поскольку буфер достаточно велик, у оператора довольно много времени, чтобы исправить ситуацию и позволить медленному потребителю сократить отставание прежде, чем он начнет пропускать сообщения.

(4) хранить вечно

лог - (по смыслу самого слова) показывать полную историю всего что когда либо происходило (и время как индекс по которому можно быстро событие на ленте времени)

- поэтому лог называют single source of truth

- The event log is not just a database of the current state like traditional SQL databases, but a database of everything that has ever happened in the system, its full history. Here, time is a natural index, making it possible for you to travel back and replay scenarios for debugging purposes, auditing, replication, failover, and so on

если много OLTP обновлений в compacted log то сжатие может не справится

- Насколько реально хранить неизменяемую историю всех изменений постоянно?
 - Ответ зависит от количества перезаписей в наборе данных.
 - Одни приложения главным образом добавляют данные и редко обновляют или удаляют их; такие данные легко сделать неизменными.
 - Другие приложения имеют высокую долю обновлений и удалений на сравнительно небольшом наборе; в этих случаях неизменяемая история способна быстро разрастаться, фрагментация может стать проблемой, а скорость сжатия и сборки мусора становится решающей для оперативной устойчивости
-
- Кроме производительности, бывают и иные обстоятельства, при которых необходимо удалить данные из соображений администрирования, несмотря на принцип неизменяемости

(5) хранить стока пока кольцевой буфер не заполнится

концептуально кафка это **circular buffer or ring buffer** (те не бесконечный)

- Такой буфер также называют круговым или кольцевым. Однако вследствие своего нахождения на диске он может быть довольно большим
- журналирование — вариант буферизации с большим буфером фиксированного размера (ограниченным доступной емкостью диска).

способен служить буфером для сообщений за **11 часов**, после чего начнет перезаписывать старые сообщения.

- Сделаем приблизительный подсчет. На момент написания этой книги емкость типичного большого жесткого диска составляла 6 Тбайт, а скорость последовательной записи — 150 Мбайт/с.
- Если записывать сообщения с максимально возможной скоростью, то диск заполнится примерно через 11 часов.
- Таким образом, он способен служить буфером для сообщений за 11 часов, после чего начнет перезаписывать старые сообщения. При использовании нескольких дисков и машин это соотношение не изменится.

На практике редко прибегают к полной скорости записи диска, так что журнал обычно способен хранить буфер за **несколько дней** или даже недель

- Независимо от того, как долго хранятся сообщения, скорость записи журнала остается более или менее постоянной, поскольку все сообщения в любом случае записываются на диск

(6) log compactiom позволяет использовать брокер сообщений для долговременного хранения сообщений, а не только для передачи при их обмене

- Уплотнение журнала, как было показано в одноименном пункте подраздела «Перехват изменений данных» текущего раздела, является одним из способов преодоления различия между журналом и состоянием базы данных: в уплотненном журнале хранятся только последние версии всех записей, а перезаписанные отбрасываются.

VARIOUS CONSISTENCY

21 декабря 2020 г. 22:18

eventual consistency различные типы согласованности в одной и той же системе

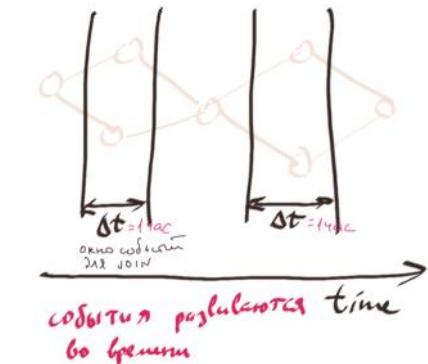
Одним из важных последствий отправки данных во множество различных сервисов является то, что мы не можем управлять согласованностью одним и тем же способом. У нас будет много копий одних и тех же данных, встроенных в разные службы, которые, если бы они были доступны для записи, могли бы привести к конфликтам и несогласованности.

Поскольку это понятие «конечной согласованности» часто нежелательно в бизнес-приложениях, одним из решений является изолирование проблем согласованности (то есть операций записи) с помощью принципа единой записи (Single Writer Principle). Например, Служба заказов будет владеть тем, как Заказ **будет развиваться со временем**. Затем каждая нисходящая служба подписывается на строго упорядоченный поток событий, создаваемых этой службой, которые они наблюдают **со своей временной точки зрения**.

Это добавляет в систему значительную «слабину», отделяя сервисы друг от друга во времени и облегчая их независимое масштабирование и развитие.

- временные окна различных событий одного и того же БП (например оформления заказа)
- зная что **благодаря eventual-consistency** события одного заказа будут где-то в пределах часового окна нам не нужно **для джойна событий** перелапывать все события с начала времен, а **сдвинуть в окне window=1час**

Воспользуемся несколькими вариантами на примере электронной почты. Представьте, что вы хотите отправить электронное письмо с подтверждением оплаты нового заказа. Мы знаем, что заказ и соответствующий ему платеж появляются примерно в одно и то же время, но мы не знаем наверняка, что будет раньше и насколько далеко они могут быть друг от друга. Однако мы можем установить для этого верхний предел - скажем, час на всякий случай.



Самое интересное в том, что эти события - гораздо больше, чем просто уведомления. События - это факты, которые развиваются и рассказывают историю. Повествование, описывающее эволюцию вашей системы в целом; день в жизни вашего бизнеса, день за днем, навсегда.

пример бизнесового окна

- Открытие на территории магазинов кафе было очень удачным решением для компании ZMart, и этот успех она хотела бы закрепить. Поэтому было решено запустить новую программу. ZMart хотела бы поддерживать посещаемость покупателями магазинов электроники за счет предложения купонов на посещение кафе (в надежде, что повышение посещаемости приведет к дополнительным продажам).
- Компания ZMart хотела бы идентифицировать покупателей, которые заказали кофе и совершили затем покупку в магазине электроники, и отправлять им купон почты сразу после этой второй транзакции (рис. 4.12). ZMart хочет выяснить, удастся ли им выработать у своих покупателей своеобразный условный рефлекс.
- Чтобы определить, когда выдавать купон, необходимо соединить информацию о заказах в кафе с информацией о продажах в магазине электроники. Код соединения потоков данных довольно прост. Начнем с подготовки данных, которые необходимо обработать для выполнения соединения.
- Мы указали, что события покупки должны отстоять друг от друга не более чем на 20 минут, но это не подразумевает какой-либо упорядоченности. Соединение будет выполнять для любых меток даты/времени, расположенных в пределах 20 минут друг от друга, независимо от порядка.
- Мы сформировали соединенный поток данных: покупка электроники, произведенная в пределах 20 минут от заказа кофе, приведет к выдаче покупателю купона на получение при следующем посещении им ZMart бесплатного кофе



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

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

асинхронная связь часто означает длительное время отклика

- Ошибки, из-за которых одна система за другой ломается, как домино, гораздо менее вероятны при использовании асинхронной связи. Системы вынуждены иметь дело с длительным временем отклика, поскольку асинхронная связь часто означает длительное время отклика

асинхронной коммуникации, которая упрощает устойчивость/resilience

- В случае сбоя микросервиса сообщение будет передано позже, но отказавший микросервис не приведет к сбою другого микросервиса. Кроме того, в главах, посвященных синхронной связи, показано, как системы могут быть устойчивыми даже при использовании синхронной связи

переосмыслить микросервисы как каскад уведомлений , отделяющий каждый источник событий от его последствий.

- we can rethink our services not simply as a mesh of remote requests and responses—where services call each other for information or tell each other what to do—but as a cascade of notifications, decoupling each event source from its consequences. мы можем переосмыслить наши сервисы не просто как сеть удаленных запросов и ответов, где сервисы обращаются друг к другу для получения информации или сообщают друг другу, что делать, но как каскад уведомлений , отделяющий каждый источник событий от его последствий.
- те потоковая обработка данных (streaming platform)
- Системы, построенные таким образом, в реальном мире бывают разных обличий. Они могут быть **мелкозернистыми и быстро выполняемыми, выполняемыми в контексте HTTP-запроса, или сложными и длительными (БП)**, манипулирующими потоком событий, которые отображают бизнес-процесс всей компании. В этом посте основное внимание уделяется первому

decoupling systems in time благодаря событиям

- Методы, основанные на событиях, меняют это положение, разделяя системы во времени и позволяя им развиваться независимо друг от друга

centralized approach to model causality of facts is event logging

- Централизованный подход к моделированию **причинно-следственной связи фактов** - это регистрация событий (подробно обсуждается в ближайшее время), тогда как децентрализованный подход заключается в использовании векторных часов или CRDT.

think and design in terms of consistency boundaries for the services

- (Pat Helland's paper, "Data on the Outside versus Data on the Inside",)
- (1) **strongly consistent**/unit of consistency / строго согласованный набор данных - относится к одному DDD агрегату
- (2) За пределами границ согласованности агрегата у нас нет другого выбора, кроме как полагаться на **eventually consistency** (конечную согласованность) (те это для событий между агрегатами)

Я склонен думать и проектировать с точки зрения **consistency boundaries** для сервисов:

1. Не поддавайтесь желанию начать с размышлений о **поведении** службы.
2. Начните с данных - фактов - и подумайте о том, как они связаны и какие зависимости имеют.
3. Определите и смоделируйте ограничения целостности и то, что необходимо гарантировать, с точки зрения предметной области и бизнеса. В этом процессе очень важно проводить интервью с экспертами в предметной области и заинтересованными сторонами.
4. Начните с нулевых гарантий для наименьшего возможного набора данных. Затем добавьте самый

слабый уровень гарантии, который решит вашу проблему, при этом пытаясь сохранить размер набора данных до минимума.

5. Пусть принцип единой ответственности ([обсуждается в разделе «Единственная ответственность» на стр. 2](#)) будет руководящим принципом.

Цель состоит в том, чтобы попытаться минимизировать набор данных, который должен быть строго согласованным. После того как вы определили существенный набор данных для Сервисов, затем обратиться к поведению и протоколам для экспонирования данных в процессе взаимодействия с другими службами и системами, определяя наши **unit of consistency (DDD агрегаты)**

- Агрегаты, которые не связываются друг на друга напрямую, могут быть повторно разделены и перемещены в кластере для почти бесконечной масштабируемости, как обрисовал Пэт Хелланд в его влиятельной статье «Жизнь за пределами распределенных транзакций». 5
- За пределами границ согласованности агрегата у нас нет другого выбора, кроме как полагаться на **eventual consistency** (конечную согласованность). В своей книге «Реализация дизайна, основанного на домене» (Addison-Wesley) Вон Вернон предлагает практическое правило, как думать об ответственности в отношении согласованности данных. Вы должны задать себе вопрос: «А чья работа обеспечивать согласованность данных?» Если ответ таков, что бизнес-логику выполняет сервис, подтвердите, что это можно сделать за один раз.

после рассмотрения DDD предметной области можно принять решение о том какие задержки допустимы

В общем, важно сделать шаг назад после многих лет предвзятых знаний и предубеждений и посмотреть, как на самом деле устроен мир. Мир редко бывает строго согласованным, и охват реальности и фактической семантики предметной области часто открывает возможности для ослабления требований согласованности.

Pushing data forward ([важно чтобы сообщения текли только в одном направлении](#)) (как водопад)

- [ненужны подтверждения, ненужны обратные сообщения](#)

- Самый быстрый способ передачи сообщения от Алисы через Боба к Чарли - это если каждая станция на пути отправляет сообщение, как только станция его получает. Единственные задержки в этом процессе связаны с передачей сообщения между станциями и обработкой сообщения на каждой станции.
- когда дело доходит до проектирования потока сообщений через реактивное приложение, **важно, чтобы пути были короткими, а сообщения текли в одном направлении, насколько это возможно**, всегда в направлении логического пункта назначения данных.
- Таким образом, количество сообщений, которыми обмениваются, сводится к минимуму, а данные обрабатываются, пока они «горячие», что означает как их актуальность для пользователя, так и их нахождение в активной памяти задействованных компьютеров.

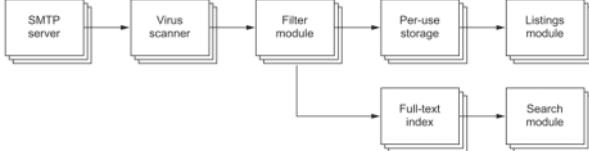
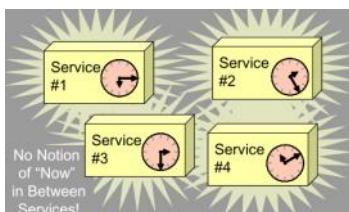


Figure 10.1. Data flows forward from the source (the SMTP server module) toward the destination, feeding to the indexing service in parallel to storing the raw data.

inside data = now | outside data = then

- **sense of "now"** for operations against inside data. However, it is different **for data on the outside of the service**. The fact that it is **unlocked** means that **the data is no longer in the "now"**. Однако для данных за пределами службы все иначе. Тот факт, что он разблокирован, означает, что данных больше нет в «сейчас». Кроме того, операторы - это запросы на операции, которые еще не произошли и которые действительно существуют в будущем (при условии, что они будут реализовываться) Наконец, мы принимаем во внимание тот факт, что разные сервисы живут в своих собственных частных временных доменах, и что это неотъемлемая часть сервис-ориентированной архитектуры
- К тому времени, как вы увидите далекий объект, он может измениться! К тому времени, когда вы увидите сообщение, данные могли измениться!
- Внутри транзакции все происходит одновременно Одновременность существует только внутри транзакции! Одновременность существует только внутри сервиса!
- У каждого микросервиса своя перспектива. Его **внутренние данные обеспечивают основу «сейчас»**. Его **внешние данные дают основу для «прошлого»**. Мое внутреннее не является вашим внутренним, так же как мое внешнее не является вашим внешним.
- Операнды могут жить либо в прошлом, либо в будущем в зависимости от модели их использования. Они живут прошлым, если у них есть копии разблокированной информации из удаленной службы. Они живут в будущем, если содержат предлагаемые значения, которые, надеюсь, будут использоваться, если оператор будет успешно завершен.
- Данные могут быть неизменными. После записи неизменяемых данных и присвоения им идентификатора содержимое данных всегда остается неизменным для этого идентификатора. Неизменяемые данные одинаковы независимо от того, **когда** на них ссылается и где они ссылаются. Неизменяемости недостаточно, чтобы избежать путаницы
- Стабильные данные имеют однозначную и неизменную интерпретацию в пространстве и времени. Слова «президент Буш» имели другое значение в 2005 году, чем в 1990 году. Эти слова нестабильны из-за отсутствия дополнительных квалифицирующих данных. Чтобы гарантировать стабильность данных, важно разрабатывать **значения, однозначные в пространстве и времени**. Отличным методом создания стабильных данных является использование отметок времени и / или управления версиями. Другой важный метод - исключить повторное использование важных идентификаторов.



event collaboration pattern (ни один микросервис не владеет всем бизнес-процесом, каждое событие микросервиса запускает один следующий шаг)

- Это позволяет набору сервисов взаимодействовать в рамках единого бизнес-процесса, при этом каждая служба вносит свой вклад, прослушивая события, а затем создавая новые. Так, например, мы можем начать с создания заказа, а затем различные службы будут развивать рабочий процесс до тех пор, пока купленный товар не дойдет до двери пользователя
- особенность Event Collaboration заключается в том, что **ни одна служба не владеет всем процессом; вместо этого каждой службе принадлежит небольшая часть** - некоторое подмножество переходов между состояниями - и они соединяются посредством цепочки событий. Таким образом, каждая служба выполняет свою работу, а затем вызывает событие, обозначающее, что она сделала. Если он обработает платеж, он вызовет событие «Платеж обработан». Если он подтвердит заказ, он повысит значение Order Validated и так далее. Эти события запускают следующий шаг в цепочке (который может снова запустить эту службу или, альтернативно, запустить другую службу).
- Отсутствие какой-либо одной точки централизованного управления означает, что такие системы часто называют хореографией : каждая служба обрабатывает некоторое подмножество переходов между состояниями, которые в совокупности описывают весь бизнес-процесс. Это можно противопоставить оркестровке , когда один процесс управляет всем рабочим процессом и управляет им из одного места, например, через диспетчер процессов.

баланс инкапсуляции и раскрытия данных

Фактически, некоторые из наиболее трудноразрешимых технологических проблем, с которыми сталкиваются предприятия, **возникают из-за расходящихся наборов данных**, распространяющихся от приложения к приложению.

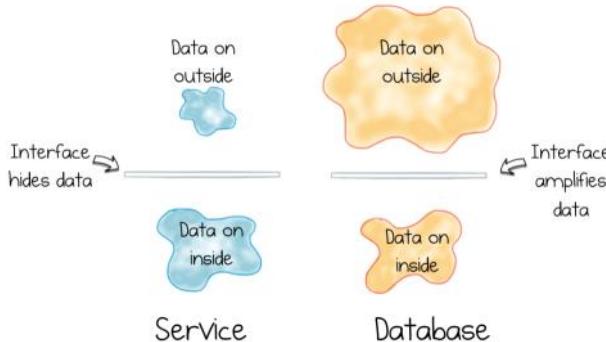
Основная проблема в том, что данные и сервисы не слишком хорошо сочетаются друг с другом. С одной стороны, **инкапсуляция побуждает нас скрывать данные; отделяют сервисы друг от друга, чтобы они могли продолжать меняться и расти. Это о планировании на будущее**. Но с другой стороны, нам нужна свобода разрезать общие данные, как и любой другой набор данных. Речь идет о том, чтобы продолжить нашу работу прямо сейчас, с теми же свободами, что и любая другая система данных.

Но системы данных не имеют ничего общего с инкапсуляцией. На самом деле, как раз наоборот. **Базы данных делают все возможное, чтобы раскрыть хранящиеся в них данные. Они поставляются с удивительно мощным декларативным интерфейсом, который может преобразовывать данные практически в любую форму, которую вы пожелаете**. Именно то, что вам нужно для исследовательского расследования, но не так хорошо, чтобы справиться с возникновением сложности в растущей сфере обслуживания

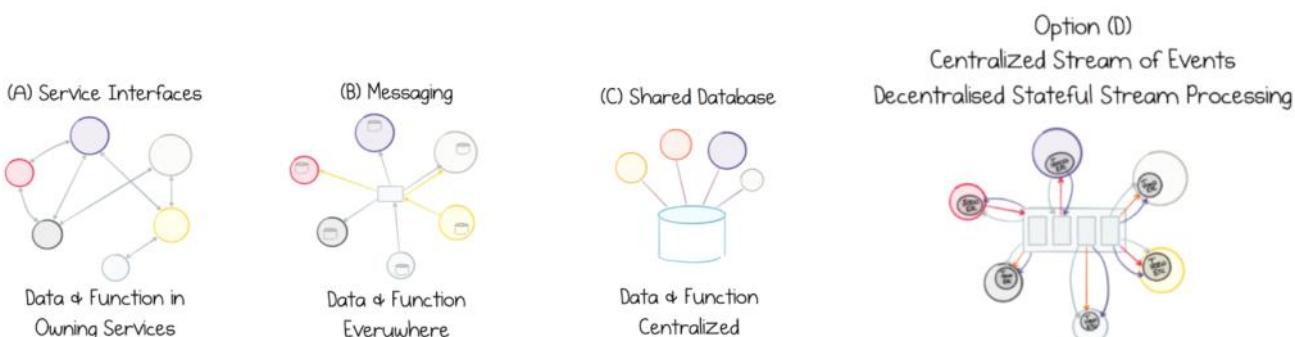
Итак, мы столкнулись с загадкой. Противоречие. Дихотомия: системы данных предназначены для раскрытия данных. Сервисы скрывают это.

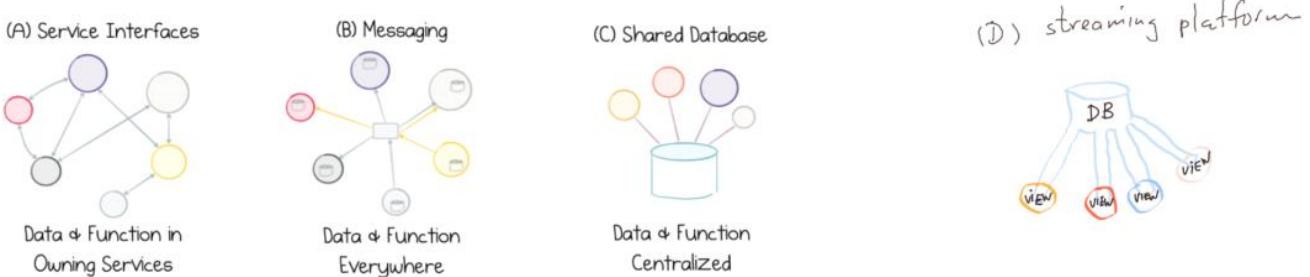
По мере развития и роста систем, основанных на услугах, мы видим, что эффекты этой дихотомии данных проявляются по-разному. Либо служебный интерфейс будет расти, открывая все больший набор функций, до такой степени, что он начинает выглядеть как какая-то странная, доморощенная база данных. Или же наступит разочарование, и мы добавим способ извлечения и перемещения целых наборов данных в массовом порядке от сервиса к сервису

Databases amplify the data they hold



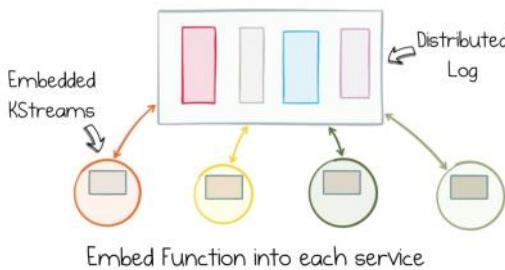
Проблема в том, что ни один из доступных сегодня подходов, сервисных интерфейсов, обмена сообщениями или общей базы данных, не дает хорошего решения для работы с внешними данными. Сервисные интерфейсы плохо подходят для обмена данными на любом уровне масштаба. Обмен сообщениями перемещает данные, но не предоставляет исторической справки, что со временем приводит к повреждению данных. Общие базы данных слишком концентрируются в одном месте, что душит прогресс. Следовательно, мы неизбежно застреваем в цикле неадекватности данных:





- Этот конкретный компромисс предполагает определенную степень централизации. Мы можем использовать для этого распределенный журнал, поскольку он обеспечивает сохраняемые масштабируемые потоки. Теперь нам нужно, чтобы наши службы могли присоединяться к этим общим потокам и работать с ними, но мы хотим избежать сложных централизованных «служб Бога», которые выполняют этот тип обработки. Таким образом, лучший подход - встроить потоковую обработку в каждую службу-потребитель. Это означает, что **сервисы могут объединять различные общие наборы данных и перебирать их в своем собственном темпе**.
- Then a stateful stream processing engine can be used to embed the declarative tools of a database right inside the consuming services. This point is important. Whilst data is stored in shared streams, which all services might access, the joins and processing a service does, is private. The smarts are isolated inside each bounded context.
- Устраним дихотомию данных, поделившись неизменяемым потоком состояния. Затем вставьте функцию в каждую службу с помощью механизма обработки потокового потока с отслеживанием состояния. Address the data dichotomy by sharing an immutable stream of state. Then push the function into each service with a Stateful Stream Processing Engine.
- Поэтому, если ваш сервис должен работать с заказами, каталогом продуктов или инвентаризацией компании, он имеет полный доступ: вы решаете, какие наборы данных следует объединить, вы решаете, где он будет выполняться, и вы решаете, когда и как его развивать с течением времени. Это означает, что **хотя сами данные являются общими, работа с этими общими данными полностью децентрализована**. Он полностью находится внутри каждой границы обслуживания, в мире, где правишь только ты. So if your service needs to operate on the company's Orders, Product Catalogue or Inventory, it has full access: you decide which datasets should be combined, you decide where it executes and you decide when and how to evolve it over time. This means that, whilst data itself is shared, operation on that shared data is fully decentralized. It sits entirely inside each service boundary, in a world where you alone reign king.

Share Data via Immutable Streams



consistency - множество типов согласованности существуют, только чтобы улучшить скорость

- Различные модели согласованности действительно отражают оптимизацию концепции упорядоченного выполнения для одной копии данных. Эти оптимизации необходимы на практике, и **большинство пользователей предпочли бы обменять немного более слабую гарантиту на лучшие характеристики производительности** (или доступности), которые обычно идут с ними. Поэтому разработчики придумывают разные способы ослабить простую гарантиту «безупречного исполнения». Эти различные оптимизации приводят к разным моделям согласованности, и поскольку существует множество параметров, которые необходимо оптимизировать, особенно в распределенных системах, существует множество результирующих моделей.
- Бизнес-системы часто НЕ нуждаются в такой работе в автономном режиме, но все же есть преимущества в том, чтобы **избежать global strong consistency и распределенных транзакций**: их сложно и дорого масштабировать, они плохо работают в разных регионах и часто относительно медленны. Фактически, опыт работы с распределенными транзакциями, охватывающими разные системы, с использованием таких методов, как XA, побудил большинство разработчиков создавать решения с учетом потребности в таких долгостоящих точках координации.
- Но с другой стороны, бизнес-системы, как правило, нуждаются в сильной согласованности**, чтобы **снизить вероятность ошибок**, поэтому есть активные сторонники, которые считают важными более высокие характеристики безопасности. Есть также аргумент в пользу того, что нужно иметь немного обоих миров. Эта золотая середина - это то место, где располагаются системы, управляемые событиями, часто с некоторой формой конечной согласованности.

важно точно знать, каковы требования, особенно в отношении согласованности, чтобы принять технически правильное решение.

- Асинхронная связь должна быть предпочтительнее синхронной связи между микросервисами из-за преимуществ, связанных с отказоустойчивостью и decoupling.
- Единственная причина возражать против этого - inconsistency.
- Поэтому важно точно знать, каковы требования, особенно в отношении согласованности, чтобы принять технически правильное решение.

важно проектировать системы, которые справляются с периодами несогласованности

- Но в мире, где приложения распределены (по географическим регионам, устройствам и т. д.), Не всегда желательно иметь единую глобальную модель согласованности. Если вы создаете данные на мобильном устройстве, они могут быть согласованы с данными на внутреннем сервере, только если они подключены. При отключении они будут, по определению, несовместимы (по крайней мере, в тот момент) и будут синхронизироваться позже, в конце концов. становится **eventually consistent**. Но **важно проектировать системы, которые справляются с периодами несогласованности**. Для мобильного устройства возможность работать в автономном режиме является желательной функцией, так же как и повторная синхронизация с внутренним сервером при повторном подключении, сходимость к согласованности при этом. Но полезность этого режима работы зависит от конкретной работы, которую необходимо выполнить. Мобильное приложение для совершения покупок может позволить вам покупать что-либо физически, пока вы снова не вернетесь в Интернет. Итак, это варианты использования, в которых глобальная сильная согласованность нежелательна.

событие можно рассматривать как своего рода репликацию данных между несколькими микросервисами.

- Однако, в отличие от полной репликации данных между узлами баз данных NoSQL, каждый микросервис может по-разному реагировать на событие и может использовать только части данных.
- Микросервис, основанный на асинхронной связи, событиях и репликации данных, соответствует системе AP (в CAP теореме).
- Возможно, микросервисы еще не получили некоторые события, поэтому данные могут быть несовместимыми. Тем не менее, система может обрабатывать запросы, используя локальные данные, и поэтому доступна даже в случае сбоя других систем.

В простейшем случае inconsistency/несоответствия исчезают, как только все события достигают всех систем

интуитивное понятие consistency (business operations execute sequentially on a single copy of data)

- intuitive notion of consistency: the idea that business operations execute sequentially on a single copy of data

[a] consequence of eventual consistency: Timeliness

для параллельно выполняемых микросервисов

- Если две службы обрабатывают один и тот же поток событий, они будут обрабатывать их с разной скоростью, поэтому одна из них может отставать от другой. Если по какой-либо причине бизнес-операция обращается к обеим службам, это может привести к несогласованности.
- Рассмотрим почтовый сервис (4) и просмотр заказов (5). Оба подписываются на один и тот же поток событий (проверенные заказы) и обрабатывают их одновременно. Выполнение в данный момент означает, что один будет немного отставать от другого. Конечно, если мы остановим запись в систему, то и представление заказов, и служба электронной почты в конечном итоге сойдутся в одном и том же состоянии, но при нормальной работе они будут занимать немного разные позиции в потоке событий.
- Таким образом, им не хватает своевременности по отношению друг к другу. Это может вызвать проблемы для пользователя, так как между службой электронной почты и службой заказов существует косвенное соединение. Если пользователь щелкает ссылку в электронном письме с подтверждением, но представление в службе заказов отстает, ссылка либо не сработает, либо вернет неправильное состояние (7).
- Таким образом, несвоевременность (например, задержка) может вызвать проблемы, если сервисы каким-либо образом связаны, но в более крупных экосистемах выгодно, чтобы сервисы были разделены, поскольку это позволяет им выполнять свою работу одновременно и изолированно, а вопросами своевременности обычно можно управлять.

если параллельное выполнение микросервисов неприемлемо, то заменим его на последовательное

- Но что, если такое поведение неприемлемо, как демонстрирует этот пример электронного письма? Что ж, мы всегда можем снова добавить последовательное выполнение. Вызов службы заказов может блокироваться до тех пор, пока не будет обновлено представление
- В качестве альтернативы мы могли бы заставить службу заказов вызывать событие View Updated, используемое для запуска службы электронной почты после обновления представления. Оба они синтезируют последовательное выполнение там, где это необходимо.

[c] consequence of eventual consistency: Collisions

для параллельно выполняемых микросервисов

- Если разные службы вносят изменения в один и тот же объект в одном потоке событий, если эти данные позже объединяются, например, в базе данных, некоторые изменения могут быть потеряны.
- Конфликты возникают, если две службы обновляют одну и ту же сущность одновременно. Если мы спроектируем систему для последовательного выполнения, этого не произойдет, но если мы разрешим одновременное выполнение, это возможно
- мы можем позволить службе проверки и налоговой службе выполнять одновременно, но в результате мы получим два события с важной информацией в каждом: один подтвержденный заказ и один заказ с добавленным налогом с продаж. Это означает, что для получения правильного порядка с применением как проверки, так и налога с продаж нам придется объединить эти два сообщения. (Таким образом, в этом случае слияние должно происходить как в почтовой службе, так и в представлении заказов.)

если параллельное выполнение микросервисов неприемлемо, то заменим его на последовательное

- Рассмотрим службу проверки и налоговую службу на рис. 11-2. Чтобы заставить их работать последовательно, мы можем принудительно запустить налоговую службу сначала (запрограммировав ее на реакцию на события «Запрошенный заказ»), а затем принудительно запустить службу проверки (запрограммировав службу так, чтобы она реагировала на события, которые были добавлены налогом с продаж). Это линеаризует выполнение каждого заказа и означает, что в конечном событии будет содержаться вся информация (т. е. Оно подтверждено и имеет добавленный налог с продаж). Конечно, последовательное выполнение событий таким образом увеличивает сквозную задержку.

финансовые операции лучше сделат последовательными (для перестраховки)

- В некоторых ситуациях эта возможность одновременно вносить изменения в один и тот же объект в разных процессах и объединять их позже может быть чрезвычайно мощной (например, инструмент интерактивной доски). Но в других случаях это может привести к ошибкам. Как правило, при построении бизнес-систем, особенно связанных с деньгами и т. п., мы склонны ошибаться в сторону осторожности.

CRDT conflict-free replicated data type (для безопасного мережа параллельных операций)

- Существует формальный метод слияния данных таким образом, который имеет гарантированную целостность; он называется бесконфликтным типом данных с репликацией или CRDT. CRDT существенно ограничивают операции, которые вы можете выполнять, чтобы гарантировать, что при изменении и последующем объединении данных вы не потеряете информацию. Обратной стороной является то, что диалект относительно ограничен.
- Google Docs employs a similar technique called operational transformation <http://svn.apache.org/repos/asf/incubator/wave/whitepapers/operational-transform/operational-transform.html>
 - В сценарии, в котором реплики документа не синхронизируются из-за сетевого раздела, локальные изменения по-прежнему принимаются и сохраняются как операции. Когда сетевое соединение возвращается в рабочее состояние, различные цепочки операций объединяются путем приведения их в линеаризованную последовательность

лучше полностью исключить возможность коллизий (а timeliness сделать допустимым) через single writer principle

- Хороший компромисс для крупных бизнес-систем - сохранить несвоевременность (что позволяет нам иметь множество реплик одного и того же состояния, доступного только для чтения), но полностью исключить возможность коллизий (запретив одновременные мутации). Мы делаем это, выделяя одного писателя single writer principle для каждого типа данных (темы) или, альтернативно, для каждого перехода между состояниями. Об этом и поговорим дальше.

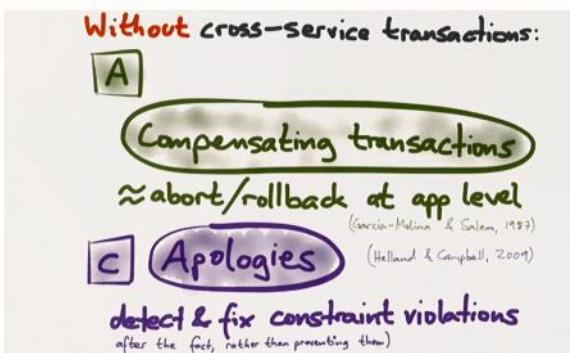
Don't Ask for Permission—Guess, Apologize, and Compensate

- So, what should we do? Let's take a step back and think about how we deal with partial and inconsistent information in real life. For example, suppose that we are chatting with a friend in a noisy bar. If we can't catch everything that our friend is saying, what do we do? We usually (hopefully) have a little bit of patience and allow ourselves to wait a while, hoping to get more information that can fill out the missing pieces. If that does not happen within our window of patience, we ask for clarification, and receive the same or additional information.
- We do not aim for guaranteed delivery of information, or assume that we can always have a complete and fully consistent set of facts. Instead, we naturally use a protocol of at-least-once message delivery and idempotent messages.
- At a very young age, we also learn how to take educated guesses based on partial information. We learn to react to missing information by trying to fill in the blanks. And if we are wrong, we take compensating actions. We need to learn to apply the same principles in system design, and rely on a protocol of: Guess; Apologize; Compensate.7 It's how the world works around us all the time.
- One example is ATMs. They allow withdrawal of money even under a network outage, taking a bet that you have sufficient funds in your account. And if the bet proved to be wrong, it will correct the account balance through a compensating action—by deducting the account to a negative balance (and in the worst case the bank will employ collection agencies to recuperate any incurred debt).
- Another example is airlines. They deliberately overbook aircrafts, taking a bet that not all passengers will show up. And if they were wrong, and all people show up, they then try to bribe themselves out of the problem by issuing vouchers—performing compensating actions.
- We need to learn to exploit reality to our advantage.

у нас есть только два варианта ответа на ошибку: компенсировать и извиниться



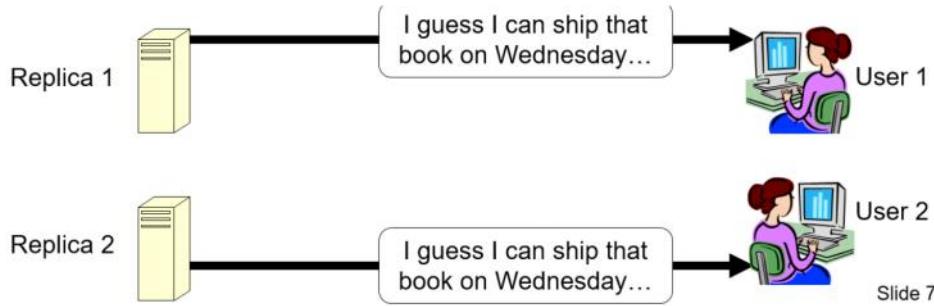
Sept 25-26, 2015
thestrangeloop.com



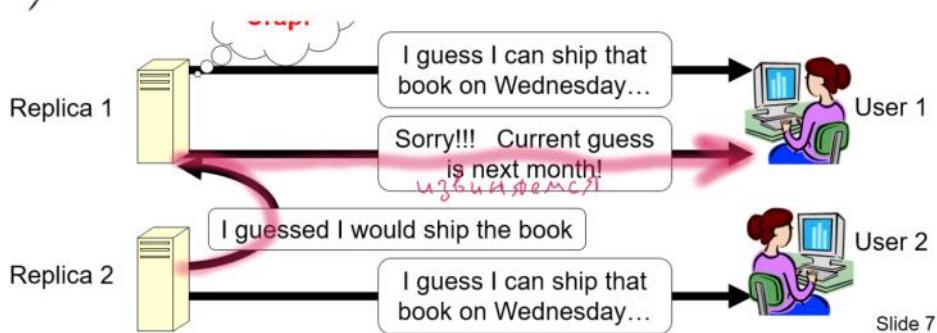
т.е я по бизнесу могу счесть конфликты допустимыми: и если конфликт произойдет с одним из людей-клиентов то я просто извинюсь перед ним или самовольно перенесу дату доставки

- <https://channel9.msdn.com/Shows/ARCast.TV/ARCastTV-Pat-Helland-on-Memories-Guesses-and-Apologies>
- <https://muckrack.com/pat-helland/articles>
- ~ Write Skew and Phantoms
- когда в CAP-теореме мы выбираем доступность

*time 1) ТАК КАК РЕПЛИК НЕ СИНХРОНИЗИРОВАНЫ ТО
ОБОИМ КОНЕЦМ ОБНОВЛЯЮТСЯ ОДНОУМ И ПУКЕ КЛАНГУ*



time 2) при синхронизации реплик обнаружили конфликт



если сеть банкомата недоступна то невыдать вообще денег приведет к раздражению клиентов и потерю прибыли (чем выдать небольшую сумму денег, которую банк готов потерять)

- В качестве примера возьмем банкомат (банкомат): когда банкомат не может получить остаток на счете клиента, есть два способа справиться с ситуацией. Банкомат мог отказать в снятии средств. Хотя это безопасный вариант, он раздражает клиента и снижает доход. В качестве альтернативы банкомат может выдать деньги - возможно, до определенного верхнего предела. Какой вариант реализовать - решение бизнеса. Кто-то должен решить, предпочтительнее ли перестраховаться, даже если это означает отказ от некоторой прибыли и раздражение клиентов, или пойти на определенный риск и, возможно, выплатить слишком много денег.

типы событий

Вот общее практическое правило: если то, что вы описываете, может быть **привязано к определенному моменту времени**, есть вероятность, что вы описываете какое-то событие, даже если для его представления требуется словесная гимнастика.

- A *description of the ongoing state of something*—The day was warm; the car was black; the API client was broken. But “the API client broke at noon on Tuesday” is an event.
- A *recurring occurrence*—The NASDAQ opened at 09:30 every day in 2018. But each individual opening of the NASDAQ in 2018 is an event.
- A *collection of individual events*—The Franco-Prussian war involved the Battle of Spicheren, the Siege of Metz, and the Battle of Sedan. But “war was declared between France and Prussia on 19 July 1870” is an event.
- A *happening that spans a time frame*—The 2018 Black Friday sale ran from 00:00:00 to 23:59:59 on November 23, 2018. But the beginning of the sale and the end of the sale are events.
- *Transactional systems*—Many of these respond to external events, such as customers placing orders or suppliers delivering parts.
- *Data warehouses*—These collect the event histories of other systems for later analysis, storing them in *fact tables*.
- *Systems monitoring*—This continually checks system- and application-level events coming from software or hardware systems to detect issues.
- *Web analytics packages*—Through these, analysts can explore website visitors’ on-site event streams to generate insights.

continuous event stream – это неограниченная последовательность отдельных событий, **упорядоченных по моменту времени**, в который произошло каждое событие.

- The start of the stream may predate our observing of the stream.
- The end of the stream is at some unknown point in the future.

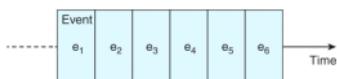


Figure 1.2 Anatomy of a continuous event stream: time is progressing left to right, and individual events are ordered within this time frame. Note that the event stream is unbounded; it can extend in both directions beyond our ability to process it.

Когда мы думаем об обработке данных, мы почти всегда думаем об обработке ограниченного набора данных поток не имеет конца , поэтому первая задача при обработке нескольких событий, чтобы придумать разумный способ ограничить поток событий.

- Когда мы думаем об обработке данных, мы почти всегда думаем об обработке ограниченного набора данных- набора точек данных, которые имеют какой-то конечный размер.

обработка непрерывного потока событий может быть помещена в дискретные окна с помощью таймера или функции пульса, которая выполняется с регулярным интервалом

- Обычно мы решаем эту проблему, применяя окно обработки к нашему потоку событий
- Рисунок 5.8 иллюстрирует эту идею разделения бесконечного потока событий на определенные окна для дальнейшей обработки

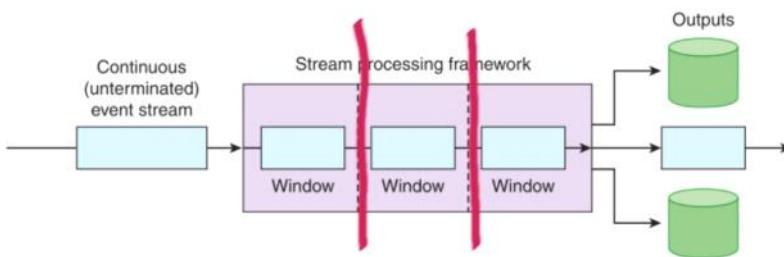


Figure 5.8 A stream processing framework typically applies windowing to a continuous (unterminated) event stream to process it and generate meaningful outputs.

- например В случае с брошенной корзиной для покупок в электронном магазине, окно обработки четко определено: мы ищем тележки для покупок, которые были оставлены их владельцем не менее 30 минут. Мы не хотим оставлять брошенную корзину покупок дольше, чем требуемые 30 минут, поэтому таймер, проверяющий каждую корзину каждые 30 секунд, был бы идеальным.

Чтобы обнаружить брошенные тележки, нам нужен способ инкапсулировать поведение покупателя в хранилище ключей

- Мы должны представить текущее состояние каждого наблюдаемого покупателя в хранилище «ключ-значение» и поддерживать это состояние в актуальном состоянии, чтобы мы могли генерировать событие « Покупатель бросает корзину», как только это поведение будет обнаружено.
- В случае хранилища «ключ-значение» от нас полностью зависит, как мы проектируем пространство ключей - значение и расположение ключей (и, следовательно, их значений) в базе данных. В этом случае мы можем использовать пару ключей с именами, чтобы поддерживать текущее состояние нашего покупателя в Samza
- <shopper>-ts should be kept up-to-date with the timestamp at which our job saw the most recent *Shopper adds item to cart* event for the given shopper.
- <shopper>-cart should be kept up-to-date with the current contents of the shopper's cart, based on aggregating all *Shopper adds item to basket* events.
- *Shopper adds item to cart*—For tracking the shopper's cart state and the time when they were last active with their cart
- *Shopper places order*—For telling us to “reset” tracking for this user

process() and window() functions work. To start with process():

- We are interested in *only Shopper places order* and *Shopper adds item to cart* events.
- When a *Shopper adds item to cart*, we update a copy of their cart stored in our key-value store and update our shopper's last active timestamp.
- When a *Shopper places order*, we delete all state about our shopper from the key-value store.

- мы просканируем все хранилище ключей и значений в поисках покупателей, которые в последний раз были активны более 30 минут назад, на основе значений <shopper>-ts . Каждый раз, когда мы его находим, мы генерируем новое событие « *Shopper abandons cart event* » , содержащее содержимое корзины для этого покупателя, полученное из <shopper>-cart

Our process() function is responsible for keeping a copy of each shopper's cart up-to-date based on their add-to-cart events; it is also responsible for understanding how recently the user added something to their cart.

Now let's briefly recap the window() function:

- Every 30 seconds, we scan the whole key-value store, looking for shoppers who were *last active* more than 30 minutes ago.
- We generate a *Shopper abandons cart* event for each shopper we find, detailing the contents of their shopping cart as recorded in our key-value store.
- We send each *Shopper abandons cart* event to our outbound Kafka stream.
- We delete from the key-value all values for the shoppers who just abandoned their carts.

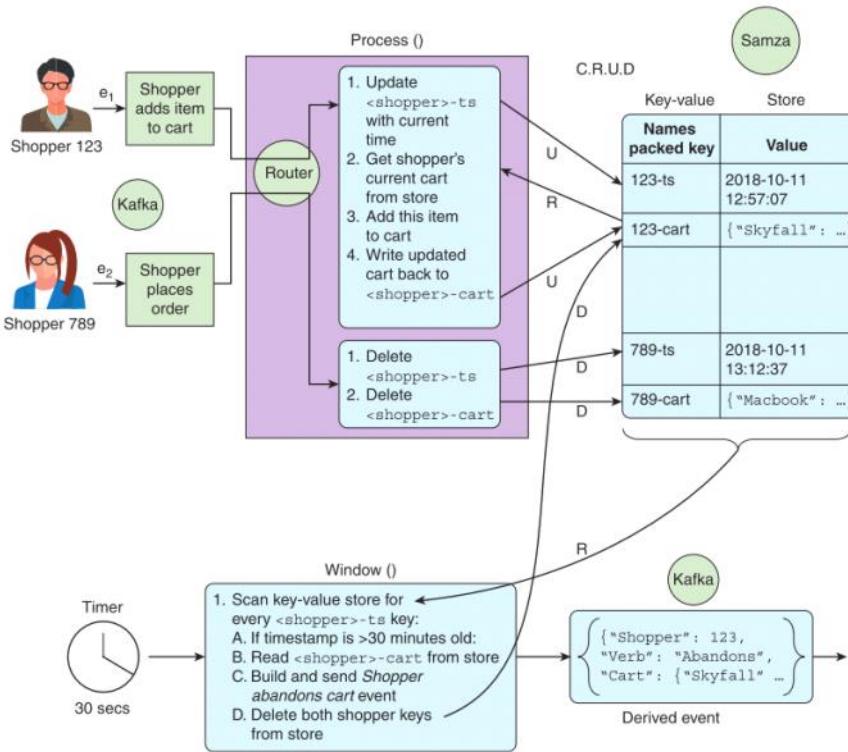


Figure 5.9 Our `process()` function parses incoming events from Nile's shoppers and updates the Samza key-value store to track the shoppers' behavior. The `window()` function then runs regularly to scan the key-value store and identify shoppers who have abandoned their carts. A *Shopper abandons cart* event is then emitted for those shoppers.

Improving our job

This Samza job is a good first stab at an abandoned cart detector. If you have time, you could enhance and extend it in a variety of ways:

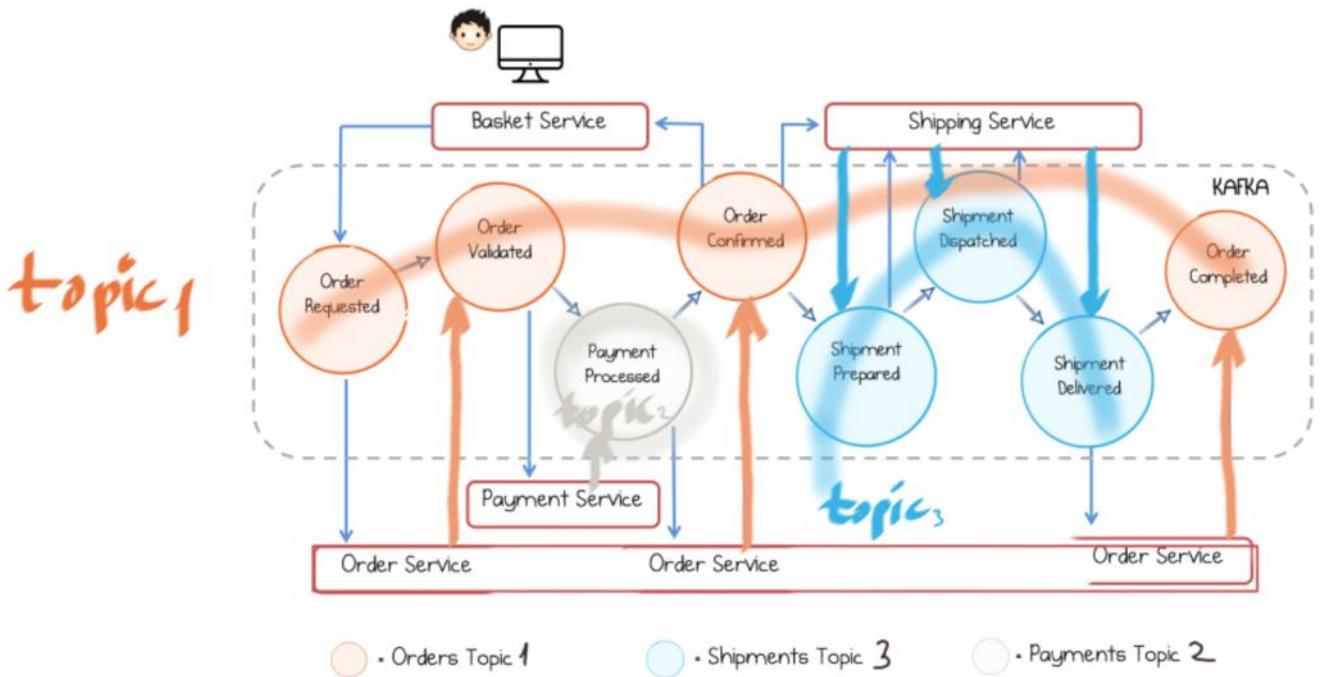
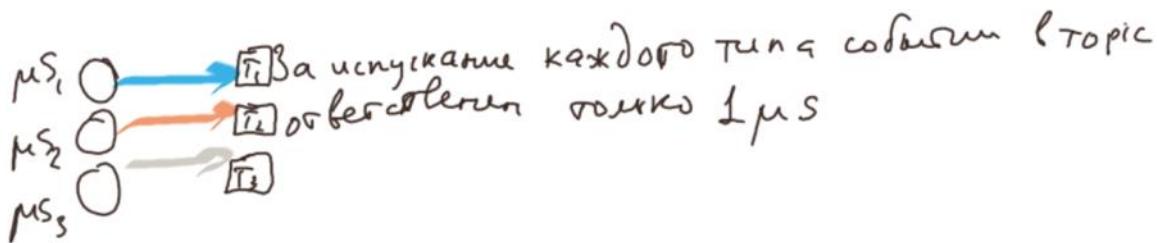
- De-duplicate items in the cart recorded in Samza's key-value store. If a shopper adds one copy of the *Skyfall* DVD to their basket twice, this is rationalized to one item with a quantity of 2.
- Use the timestamps of incoming events (from the event context) to determine when the shopper was last active, rather than basing it on the time when `process()` is running, as now.
- Define a new *Shopper removes item from cart* event, and use these events to remove items from the shopper's cart as stored in Samza's key-value store.
- Base each shopper's last-active timestamp on all events for that shopper, not just *Shopper adds item to cart* events.
- Explore more-sophisticated ways of determining whether a cart is abandoned, instead of the strict 30-minute cutoff. You could try varying the cutoff based on the shopper's previously observed behaviors.

* Single Writer Principle

4 января 2021 г. 18:49

Single Writer Principle - одним типом событий владеет/испускает только один микросервис

- **?????**те первоисточник только один, а тех кто ??обновляет событие может быть несколько микросервисов (или же все обновления срач в отдельные топики, где они потом все опять собираются микросервисом-первоисточником)
- термин от Martin Thompson (основная идея связана с actor model, single responsibility principle)
- один микросервис -> один тип событий -> один топик
- Полезный принцип, который можно применить с этим стилем системы, - это возложить ответственность за распространение событий определенного типа на одну службу: одну запись . Таким образом, служба складских запасов будет владеть тем, как будет развиваться «Инвентаризация запасов» с течением времени, служба заказов будет владеть Заказами и т.
- Это помогает обеспечить согласованность воронки, проверку и другие проблемы «пути записи» через один путь кода (хотя и не обязательно за один процесс). Итак, в приведенном ниже примере обратите внимание, что служба заказов берет на себя управление каждым изменением состояния, сделанным для заказа, но весь поток событий охватывает заказы, платежи и отгрузки, каждый из которых управляется соответствующими службами.
- Поскольку это понятие «конечной согласованности» часто нежелательно в бизнес-приложениях, одно из решений состоит в том, чтобы изолировать проблемы согласованности (то есть операции записи) с помощью принципа единой записи . Например, Служба заказов будет владеть тем, как Заказ будет развиваться во времени. Затем каждая нисходящая служба подписывается на строго упорядоченный поток событий, создаваемых этой службой, которые они наблюдают со своей временной точки зрения.
- Это добавляет значительную «слабину» в систему, отделяя сервисы друг от друга во времени и облегчая их независимое масштабирование и развитие. Мы рассмотрим пример того, как это работает на практике, позже в этой статье (служба инвентаризации), но сначала нам нужно взглянуть на механику и инструменты, используемые для сшивания этих экосистем.
- When the single writer principle is applied in conjunction with Event Collaboration , each writer evolves part of a single business workflow through a set of successive events



single responsibility principle

- ответственность за распространение событий определенного типа возлагается на один микросервис - одного писателя . Таким образом, служба инвентаризации определяет, как складские запасы продвигаются с течением времени, служба заказов отвечает за последовательность заказов и так далее.
- Conflating writes into a single service makes it easier to manage consistency efficiently
- <https://mechanical-sympathy.blogspot.com/2011/09/single-writer-principle.html>
- https://en.wikipedia.org/wiki/Actor_model
- bit.ly/deds-end-of-an-era

этот принцип позволяет избежать коллизий мержа параллельно выполняемых микросервисов (также позволяет избежать блокировок, взаимных исключений)

- см ниже "(б) consequence of eventual consistency: Collisions"
- Если у вас есть система, которая может соблюдать этот принцип, тогда каждый контекст выполнения может тратить все свое время и ресурсы на обработку логики для своей цели, и не тратить циклы и ресурсы на решение проблемы конкуренции (и блокировок)
- Принцип единственного писателя позволяет избежать этой проблемы, потому что он никогда не должен иметь дело с записью последней версии элемента данных, который мог быть записан другим потоком в настоящее время находится в буфере хранилища другого ядра.

позволяет бесконечно масштабироваться/распараллеливаться

- Вы также можете масштабировать без ограничений, пока оборудование не будет насыщено.
- Я не говорю, что блокировки и оптимистические стратегии плохи и не должны использоваться. Они отлично

подходит для многих задач. Например, начальная загрузка параллельной системы или выполнение основных этапов состояния в конфигурации или справочных данных. Однако, если основной поток транзакций воздействует на конкурирующие данные, и необходимо использовать блокировки или оптимистические стратегии, то масштабируемость фундаментально ограничена.

It allows versioning and consistency checks to be applied in a single place (те улучшает консистентность создавая local points of consistency вместо global consistency model.)

- Поскольку это понятие «конечной согласованности» часто нежелательно в бизнес-приложениях, одно из решений состоит в том, чтобы изолировать проблемы согласованности (то есть операции записи) с помощью принципа единой записи . Например, Служба заказов будет владеть тем, как Заказ будет развиваться во времени. Затем каждая нисходящая служба подписывается на строго упорядоченный поток событий, создаваемых этой службой, которые они наблюдают со своей временной точки зрения.
- так как источник событий один - то участок от микросервиса-источника до приемников - как бы однопоточен (и события последовательны) те local point of consistency: When the single writer principle is applied in conjunction with Event Collaboration (discussed in Chapter 5), each writer evolves part of a single business workflow through a set of successive events
- So, instead of sharing a global consistency model (e.g., via a database), we use the single writer principle to create local points of consistency that are connected via the event stream. There are a couple of variants on this pattern, which we will discuss in the next two sections.

It isolates the logic for evolving each business entity, in time, to a single service, making it easier to reason about and to change (так как только один микросервис владеет ивентом, то он может легко менять схему сообщений как ему надо)

-

? вариант 1) Single Writer Principle: Command Topic (command topic + entity topic)

- command topic используется для отделения начальной команды от последующих событий
 - тема команды может быть записана любым процессом и используется только для инициирующего события.
- Entity topic может быть записана только службой-владельцем: единственным писателем.
- Разделение этих двух позволяет администраторам строго применять принцип единого писателя, настраивая права доступа к темам..

Topic	OrderCommandTopic	OrdersTopic
Event types	OrderRequest(ed)	OrderValidated, OrderCompleted
Writer	Any service	Orders service

вариант 2) Single Writer Principle: Single Writer Per Transition

- Менее строгий вариант принципа единого писателя предполагает, что сервисы владеют отдельными переходами, а не всеми переходами в теме . Так, например, платежный сервис может вообще не использовать тему оплаты. Он может просто добавить дополнительную информацию об оплате к существующему сообщению о заказе (таким образом, в схеме заказа будет раздел «Платеж»). Платежная служба владеет только этим одним переходом, а служба заказов владеет остальными.

Table 11-2. The order service and payment services both write to the orders topic, but each service is responsible for a different state transition

Service	Orders service	Payment service
Topic	OrdersTopic	OrdersTopic
Writable transition	OrderRequested->OrderValidated PaymentReceived->OrderConfirmed	OrderValidated->PaymentReceived

* interactive queries functionality

4 января 2021 г. 18:46

как узнать в какой партиции кафки находится заданный ключ (kafka streams API)

Поскольку данные разделены на разделы, их можно масштабировать по горизонтали (Kafka Streams поддерживает динамическую перебалансировку нагрузки), но это также означает, что запросы GET должны направляться на правильный узел - тот, который имеет раздел для запрашиваемого ключа. Это выполняется автоматически с помощью функции интерактивных запросов в Kafka Streams. 2

возможно потребуется поиск другого стримс-клиента если данного ключа в данной партиции не оказалось

- тогда через кафка-метаданные ищем хост и перенаправляем запрос на него через http
- ?? почему бы сразу не направить на нужный хост, если мы знаем ключ и хэш функцию
- Note also that the Orders Service is partitioned over three nodes, so GET requests must be routed to the correct node to get a certain key. This is handled automatically using the Interactive Queries functionality in Kafka Streams, although the example has to implement code to expose the HTTP endpoint.

Чтобы инстансы микросервиса могли обмениваться между-собой данными по сети, вы должны добавить в свое приложение уровень удаленного вызова процедур (RPC) (например, REST API).

Procedure	Application instance	Entire application
Query local state stores of an app instance	Supported	Supported
Make an app instance discoverable to others	Supported	Supported
Discover all running app instances and their state stores	Supported	Supported
Communicate with app instances over the network (RPC)	Supported	Not supported (you must configure)

в том числе interactive queries также называется возможность напрямую обращаться к local state store

- см queriable state в [сохраняем состояние в трансформации](#)
- Он позволяет рассматривать уровень обработки потока как легкую, встроенную базу данных и напрямую запрашивать состояние вашего приложения обработки потока без необходимости материализовать это состояние во внешних базах данных или хранилище. Apache Kafka поддерживает это состояние и управляет им, а также гарантирует высокую доступность и отказоустойчивость. Таким образом, эта новая функция обеспечивает гиперконвергенцию обработки и хранения в одно простое в использовании приложение, использующее API Streams Apache Kafka.
- <https://www.confluent.io/blog/unifying-stream-processing-and-interactive-queries-in-apache-kafka/>

interactive query = materialized view

- Идея интерактивных запросов не нова; аналогичная концепция фактически возникла в традиционных базах данных, где ее часто называют «материализованными представлениями»
- Благодаря нашему опыту работы с базами данных в сочетании с нашим опытом потоковой обработки мы рассмотрели ключевой вопрос: можно ли применить концепцию материализованных представлений к современному механизму потоковой обработки для создания мощной и универсальной конструкции для создания приложений и микросервисов с отслеживанием состояния?
- наша концепция заключалась в том, чтобы вывести потоковую обработку из ниши больших

- данных и сделать ее доступной в качестве основной модели разработки приложений.
- Для нас ключом к реализации этого видения является радикальное упрощение того, как пользователи могут обрабатывать данные в любом масштабе - малом, среднем, большом - и, по сути, одна из наших мантр - «Создавайте приложения, а не кластеры!»

мы можем устраниТЬ или уменьшить потребность во внешней базе данных.

- Вот как тогда будет выглядеть приложение
- Все состояние в вашем приложении теперь можно запросить через Interactive Query API.
- RocksDB Эти встроенные базы данных действуют как материализованные представления журналов, которые хранятся в Apache Kafka
- We have made the case in the past that, for streams, materialized views can be thought as a cached subset of a log (i.e., topics in Kafka). Embedded databases and Interactive Queries are our implementations of this concept in Apache Kafka.



Interactive Queries allows developers to query a streaming app's embedded state stores

- Интерактивные запросы позволяют разработчикам запрашивать встроенные хранилища состояний потокового приложения.
- Хранилища состояний хранят последние результаты узла процессора, например узла, который агрегирует данные для вычисления суммы или среднего
- Хранилище состояний - это встроенная база данных (по умолчанию RocksDB)

Интерактивные запросы доступны только для чтения

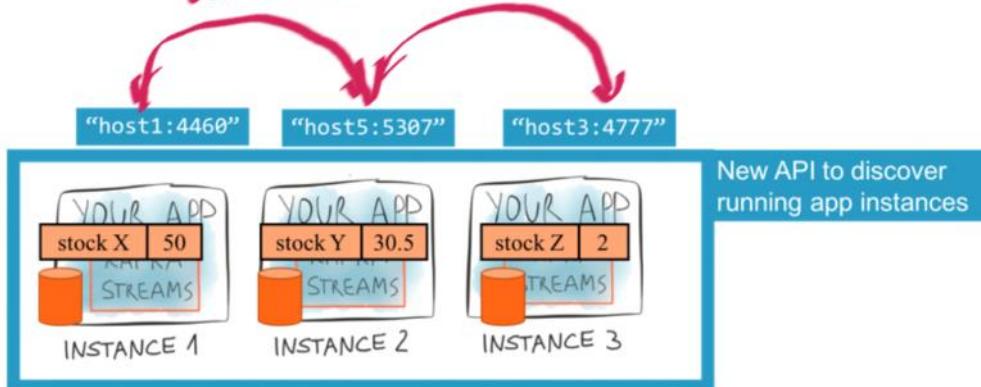
- т. Е. Никакие изменения в хранилищах состояний не допускаются.
- Это сделано для того, чтобы избежать несоответствия состояний внутренней работе приложения для обработки потоков.
- На практике разрешения только для чтения достаточно для большинства приложений, которые хотят использовать данные из запрашиваемого потокового приложения

We have made each instance aware of each other instance's state stores through periodic metadata exchanges, which we provide through Kafka's group membership protocol

- Приложение может иметь несколько запущенных экземпляров, каждый со своим собственным набором хранилищ состояний
- Мы сделали каждый экземпляр осведомленным о хранилищах состояний каждого другого экземпляра посредством периодического обмена метаданными, который мы предоставляем через протокол членства в группах Kafka.
- Начиная с Confluent Platform 3.1 и Apache Kafka 0.10.1, каждый экземпляр может предоставлять свои метаданные информации о конечной точке (имя хоста и порт, вместе известные как параметр конфигурации « application.server ») другим экземплярам того же приложения.
- Новые API интерактивных запросов позволяют разработчику получать метаданные для заданного имени и ключа хранилища и делать это в экземплярах приложения.

kafka's group membership protocol

общий список приложений



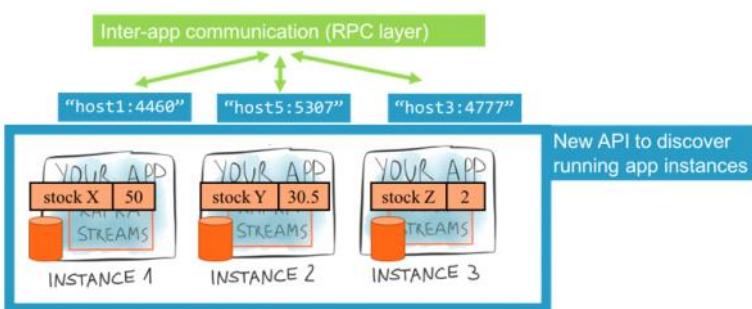
Следовательно, вы можете узнать, где находится хранилище с конкретным ключом, изучив метаданные.

- Итак, теперь мы знаем, в каком хранилище и в каком экземпляре приложения хранится наш ключ,

как на самом деле запросить это (потенциально удаленное) хранилище?

Confluent provides a reference REST-based implementation for the RPC layer for a simple reference app

- Confluent предоставляет эталонную реализацию на основе REST для уровня RPC для простого эталонного приложения (код см. В следующем разделе). Эта функция была намеренно не включена в Apache Kafka
- После создания уровня RPC любое приложение, например панель мониторинга или микросервис, написанное на Scala или Python, может интерактивно запрашивать у вашего приложения последние результаты обработки.



Получите список всех запущенных экземпляров в системе. Вот чего вам стоит ожидать:

<https://gist.github.com/miguno/be8ee6e566c4f61e516efbc3be8b4a9a>

Перечислите запущенные экземпляры, которые в настоящее время управляют «подсчетом слов» государственного хранилища: <https://gist.github.com/miguno/aaa8bec8516dcf10ce8abb16fc0c689>

Запросите пять лучших песен: <https://gist.github.com/miguno/f37285ee905623893e54aa706f8c72f7>

Запросите пятерку лучших песен по жанрам:

<https://gist.github.com/miguno/5910d4f123ab0461cd4eb61d89cd58d8>

Запросить подробную информацию о конкретной песне:

<https://gist.github.com/miguno/39397d097a7f57e42ffafc9523c5cb47>

пример 1

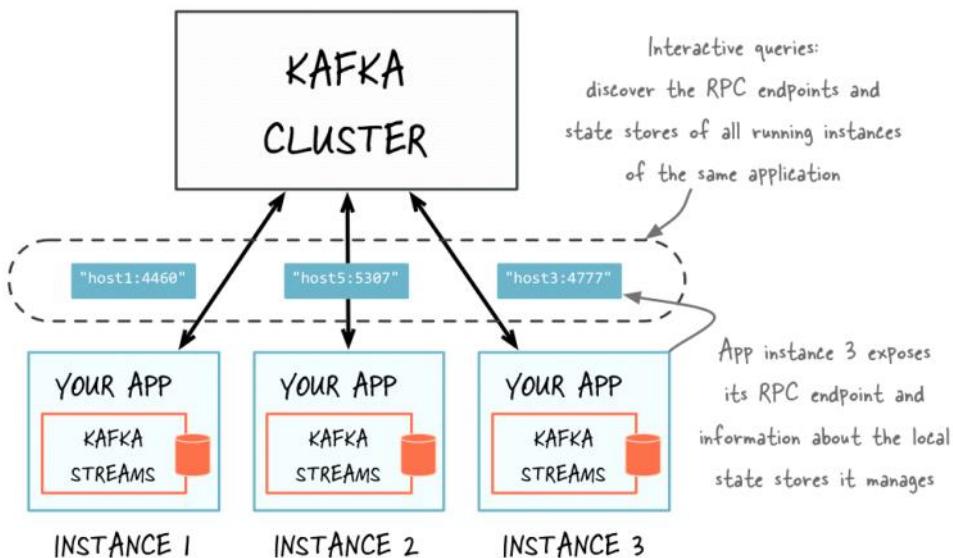
Каждый экземпляр приложения может напрямую запрашивать любое из своих локальных хранилищ состояний любого инстанса.

<https://kafka.apache.org/10/documentationstreams/developer-guide/interactive-queries.html>

Чтобы запросить удаленные состояния для всего приложения, вы должны предоставить полное состояние приложения другим приложениям, включая приложения, работающие на разных машинах.

Например, у вас есть приложение Kafka Streams, которое обрабатывает пользовательские события в многопользовательской видеоигре, и вы хотите получить последний статус каждого пользователя напрямую и отобразить его в мобильном приложении. Вот необходимые шаги, чтобы сделать полное состояние вашего приложения доступным для запросов:

- Добавьте в свое приложение уровень RPC, чтобы с его экземплярами можно было взаимодействовать через сеть (например, REST API, Thrift, настраиваемый протокол и т. д.). Экземпляры должны отвечать на интерактивные запросы. Для начала вы можете следовать приведенным справочным примерам.
- Предоставьте доступ к конечным точкам RPC экземпляров вашего приложения с помощью параметра `application.server` конфигурации Kafka Streams. Поскольку конечные точки RPC должны быть уникальными в сети, каждый экземпляр имеет собственное значение для этого параметра конфигурации. Это делает экземпляр приложения доступным для обнаружения другими экземплярами.
- На уровне RPC обнаруживайте удаленные экземпляры приложений и их хранилища состояний и запрашивайте локально доступные хранилища состояний, чтобы сделать полное состояние вашего приложения доступным для запросов. Экземпляры удаленного приложения могут пересыпать запросы другим экземплярам приложения, если конкретному экземпляру не хватает локальных данных для ответа на запрос. Локально доступные хранилища состояний могут напрямую отвечать на запросы.



в конкретном экземпляре стримса при запуске стримса в его конфиг мы должны передать host:port этого конкретного инстанса `application.server` пропертя кафа стримса

- Значение этого свойства конфигурации будет различаться в разных экземплярах вашего приложения.
- Когда это свойство установлено, Kafka Streams будет отслеживать информацию о конечной точке RPC для каждого экземпляра приложения, его хранилищ состояния и назначенных потоковых разделов через экземпляры `StreamsMetadata`.

```
final KafkaStreams streams = new KafkaStreams( createOrdersMaterializedView().build(), config( bootstrapServers, defaultConfig ) );  
private Properties config( final String bootstrapServers,
```

```

final Properties defaultConfig )
{
    final Properties props = baseStreamsConfig( bootstrapServers, "/tmp/kafka-streams", SERVICE_APP_ID, defaultConfig );
    props.put( StreamsConfig.APPLICATION_SERVER_CONFIG, host + ":" + port );
    return props;
}

```

теперь найдем в каком инстансе заданный ключ

```

private HostStoreInfo getHostForOrderId( final String orderId )
{
    return metadataService.streamsMetadataForStoreAndKey( ORDERS_STORE_NAME, orderId, Serdes.String()
        .serializer() );
}

/**
 * Use Kafka Streams' Queryable State API to work out if a key/value pair is located on
 * this node, or on another Kafka Streams node. This returned HostStoreInfo can be used
 * to redirect an HTTP request to the node that has the data.
 */
private HostStoreInfo getKeyLocationOrBlock( final String id,
                                             final AsyncResponse asyncResponse )
{
    HostStoreInfo locationOfKey;
    while( locationMetadataIsUnavailable( locationOfKey = getHostForOrderId( id ) ) )
    {
        //The metastore is not available. This can happen on startup/rebalance.
        if( asyncResponse.isDone() )
        {
            //The response timed out so return
            return null;
        }
        try
        {
            //Sleep a bit until metadata becomes available
            Thread.sleep( Math.min( Long.parseLong( CALL_TIMEOUT ), 200 ) );
        }
        catch( final InterruptedException e )
        {
            e.printStackTrace();
        }
    }
    return locationOfKey;
}

private boolean locationMetadataIsUnavailable( final HostStoreInfo hostWithKey )
{
    return NOT_AVAILABLE.host()
        .equals( hostWithKey.getHost() ) && NOT_AVAILABLE.port() == hostWithKey.getPort();
}

/**
 * Perform a "Long-Poll" styled get. This method will attempt to get the value for the passed key
 * blocking until the key is available or passed timeout is reached. Non-blocking IO is used to
 * implement this, but the API will block the calling thread if no metastore data is available
 * (for example on startup or during a rebalance)
 *
 * @param id
 * - the key of the value to retrieve
 * @param timeout
 * - the timeout for the long-poll
 * @param asyncResponse
 * - async response used to trigger the poll early should the appropriate
 * value become available
 */
@GET
@ManagedAsync
@Path( "/orders/{id}" )
@Produces( { MediaType.APPLICATION_JSON,
            MediaType.TEXT_PLAIN } )
public void getWithTimeout( @PathParam( "id" ) final String id,
                           @QueryParam( "timeout" ) @DefaultValue( CALL_TIMEOUT ) final Long timeout,
                           @Suspended final AsyncResponse asyncResponse )
{
    setTimeout( timeout, asyncResponse );

    final HostStoreInfo hostForKey = getKeyLocationOrBlock( id, asyncResponse );

    if( hostForKey == null )
    {
        //request timed out so return
        return;
    }
    //Retrieve the order Locally or reach out to a different instance if the required partition is hosted elsewhere.
    if( thisHost( hostForKey ) )

```

```

        {
            fetchLocal( id, asyncResponse, ( k, v ) -> true );
        }
    else
    {
        final String path = new Paths( hostForKey.getHost(), hostForKey.getPort() ).urlGet( id );
        fetchFromOtherHost( path, asyncResponse, timeout );
    }
}

private void fetchFromOtherHost( final String path,
final AsyncResponse asyncResponse,
final long timeout )
{
    log.info( "Chaining GET to a different instance: " + path );
    try
    {
        final OrderBean bean = client.target( path )
            .queryParam( "timeout", timeout )
            .request( MediaType.APPLICATION_JSON_TYPE )
            .get( new GenericType<OrderBean>()
            {
            } );
        asyncResponse.resume( bean );
    }
    catch( final Exception swallowed )
    {
        log.warn( "GET failed.", swallowed );
    }
}

private final Client client = ClientBuilder.newBuilder()
    .register( JacksonFeature.class )
    .build();

```

пример 2

Настройка и обнаружение распределенного хранилища состояния

- При получении экземпляром Kafka Streams запроса для заданного ключа необходимо выяснить, присутствует ли этот ключ в локальном хранилище. А главное, если ключ там отсутствует (не локальный), то необходимо узнать, в каком экземпляре находится данный ключ, и выполнить запрос к соответствующему хранилищу

Возвращаясь к примеру предоставления доступа к хранилищам состояния для выполнения запросов, можно видеть, что если сделать хранилище состояния на экземпляре A доступным веб-сервису или другой внешней утилите для запросов, то из него можно будет извлечь только значение для ключа "Energy".

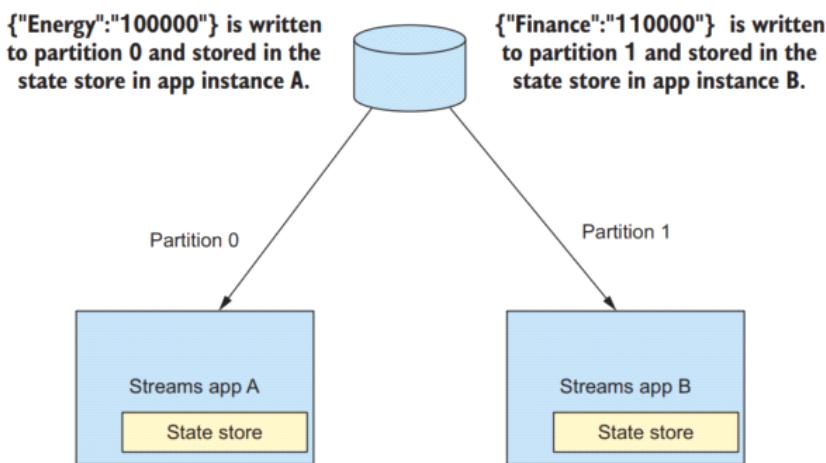


Figure 9.8 Key and value distribution in state stores

Экземпляр A не содержит ключа "Finance", но он обнаруживает, что искомый ключ есть в экземпляре Б.

Поэтому экземпляр A (те программист должен явно написать этот код) обращается к встраиваемому серверу на экземпляре Б, который извлекает нужные данные и возвращает результат исходной запрашивающей стороне.



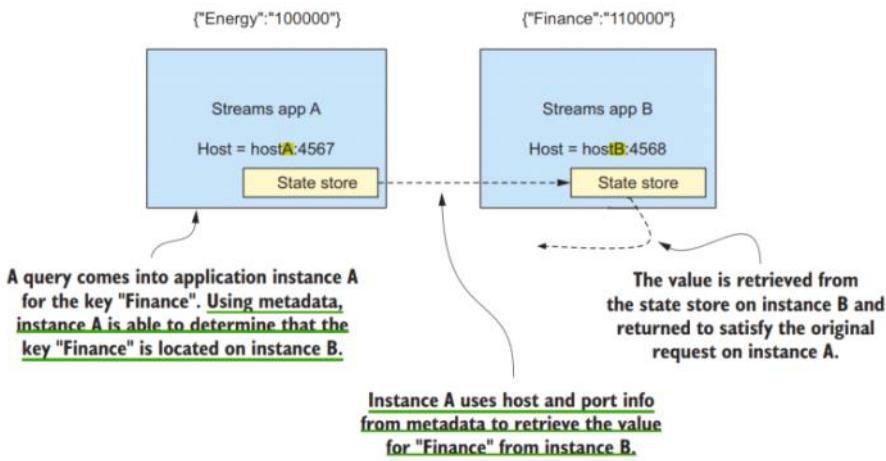


Figure 9.9 Key and value query-and-discovery process

установка конфига клиента

- Для активизации возможности выполнения интерактивных запросов необходимо задать параметр StreamsConfig.APPLICATION_SERVER_CONFIG. Его значение должно состоять из имени хоста приложения Kafka Streams и порта, на котором прослушивается сервис запросов, в формате имя_хоста:порт
- На отдельном узле интерактивные запросы будут работать без каких-либо дополнительных усилий, но готовый механизм RPC отсутствует — вам придется реализовать свой собственный. В этом разделе приведено одно из возможных решений, но вы можете реализовать свое, и я уверен, что многие из вас сумеют найти нечто получше моего варианта. Отличный пример еще одной реализации RPC можно найти в GitHub-репозитории Confluent kafkastreams-examples: <http://mng.bz/Ogo3>.
- необходимо передавать два аргумента при запуске приложения Kafka Streams: имя хоста и порт, на котором будет слушать встраиваемый сервис

Listing 9.6 Setting the hostname and port

```
public static void main(String[] args) throws Exception {
    if(args.length < 2){
        LOG.error("Need to specify host and port");
        System.exit(1);
    }
    String host = args[0];
    int port = Integer.parseInt(args[1]);
    final HostInfo hostInfo = new HostInfo(host, port); ← Creates a HostInfo object for later use in the application
    Properties properties = getProperties();
    properties.put(
        StreamsConfig.APPLICATION_SERVER_CONFIG, host+":"+port); ← Sets the config for enabling interactive queries
    // other details left out for clarity
}
```

создаем веб сервера в каждом микросервисе, для запросов микросервисов друг к другу

- В этом коде мы создали экземпляр InteractiveQueryServer — класса-адаптера для веб-сервера Spark и написали код для управления вызовами веб-сервисов, а также запуска и останова веб-сервера.
- В главе 7 мы обсуждали использование интерфейса StateListener для уведомления о различных состояниях приложения Kafka Streams. А здесь демонстрируется его эффективное применение. Напомню, что при выполнении интерактивного запроса необходимо с помощью экземпляра StreamsMetadata выяснить, являются ли данные локальными по отношению к обрабатывающему запрос экземпляру приложения. Мы установили состояние сервера запросов в true, тем самым разрешив доступ к метаданным только в том случае, если приложение находится в состоянии выполнения
- Главное — помнить, что возвращаемые метаданные отображают мгновенное состояние структуры приложения Kafka Streams. В любой момент вам может понадобиться масштабировать приложение в сторону расширения или наоборот. В подобном случае (или в случае любого другого подходящего под требования события, например добавления топиков с узлом-источником на основе регулярного выражения) приложение Kafka Streams проходит через процесс перебалансировки, в результате чего распределение секций может поменяться. В данном случае применение запросов разрешается только в состоянии выполнения, но вы можете воспользоваться другой стратегией, если счтете ее подходящей для своих целей.
- Далее следует еще один пример идеи, описанной в главе 7: описание обработчика для неперехваченных исключений (UncaughtExceptionHandler). В данном случае мы заносим ошибку в журнал, а затем останавливаем приложение и сервер запросов. Поскольку это приложение работает в течение неограниченного времени, мы добавили еще точку подключения для останова всех компонентов после завершения вами работы демонстрационного приложения.

Listing 9.7 Initializing the web server and setting its status

```
 Adds a StateListener to only enable
 queries to state stores until ready
 // details left out for clarity

 KafkaStreams kafkaStreams = new KafkaStreams(builder.build(), streamsConfig);
 InteractiveQueryServer queryServer =
 => new InteractiveQueryServer(kafkaStreams, hostInfo);
 queryServer.init();

 kafkaStreams.setStateListener((newState, oldState) -> {
 if (newState == KafkaStreams.State.RUNNING && oldState ==
 => KafkaStreams.State.REBALANCING) {
 LOG.info("Setting the query server to ready");
 queryServer.setReady(true);
 } else if (newState != KafkaStreams.State.RUNNING) {
 LOG.info("State not RUNNING, disabling the query server");
 queryServer.setReady(false);
 }
 });

 kafkaStreams.setUncaughtExceptionHandler((t, e) -> {
 LOG.error("Thread {} had a fatal error {}", t, e, e);
 shutdown(kafkaStreams, queryServer);
 });

 Runtime.getRuntime().addShutdownHook(new Thread(() -> {
 shutdown(kafkaStreams, queryServer);
 }));

```

Creates the embedded web server (actually a wrapper class)

Enables queries to state stores once the Kafka Streams application is in a RUNNING state. Queries are disabled if the state isn't RUNNING.

Sets an Uncaught-ExceptionHandler to log unexpected errors and close everything down

Adds a shutdown hook to close everything down when the application exits normally

в веб сервере - поставить в соответствие URL-путем правильные методы для выполнения

- Данные отображения довольно просты, но обратите внимание, что соответствие операции извлечения из оконного хранилища задается дважды. Для извлечения значений из оконного хранилища необходимо указать, от какого момента времени и до какого.
- обратите внимание на проверку булева значения ready при задании соответствий. Указанное значение задается в StateListener. Если значение ready равно false, то программа не будет пытаться обработать запрос и вернет сообщение, что хранилища в настоящий момент недоступны. Это логично, поскольку оконное хранилище сегментировано по времени, причем размер сегментов задается при создании хранилища (мы обсуждали оконные операции в подразделе 5.3.2). Но я здесь немного жульничую и предлагаю вам взглянуть в следующем примере на метод, принимающий в качестве параметров только ключ и хранилище со значениями по умолчанию для временных параметров «от» и «до».

Listing 9.8 Mapping URL paths to methods

```
public void init() {
 LOG.info("Started the Interactive Query Web server");

 get("/kv/:store", (req, res) -> ready ?
 => fetchAllFromKeyValueStore(req.params()) :
 => STORES_NOT_ACCESSIBLE);
 get("/session/:store/:key", (req, res) -> ready ?
 => fetchFromSessionStore(req.params()) :
 => STORES_NOT_ACCESSIBLE);
 get("/window/:store/:key", (req, res) -> ready ?
 => fetchFromWindowStore(req.params()) :
 => STORES_NOT_ACCESSIBLE);
 get("/window/:store/:key/:from/:to", (req, res) -> ready ?
 => fetchFromWindowStore(req.params()) :
 => STORES_NOT_ACCESSIBLE);
}
```

Mapping to retrieve all values from a plain key/value store

Mapping to return all sessions (from a session store) for a given key

Mapping for a window store with no times specified

Mapping for a window store with from and to times

как найти в какой из баз БД микросервисов находится данный ключ

В классе KafkaStreams есть несколько методов, с помощью которых можно извлекать информацию из всех работающих в данный момент экземпляров с одинаковым идентификатором приложения и задавать значение параметра APPLICATION_SERVER_CONFIG. В табл. 9.1 приведены названия и описания этих методов.

Для получения информации обо всех экземплярах, доступных для интерактивных запросов, можно использовать метод KafkaStreams.allMetadata. При написании интерактивных запросов я чаще всего применяю метод KafkaStreams.allMetadataForKey.

Table 9.1 Methods for retrieving store metadata

Name	Parameter(s)	Usage
allMetadata	N/A	All instances, some possibly remote
allMetadataForStore	Store name	All instances (some remote) containing the named store
allMetadataForKey	Key, Serializer	All instances (some remote) with the store containing the key
allMetadataForKey	Key, StreamPartitioner	All instances (some remote) with the store containing the key

- Данный запрос отображается на метод fetchFromWindowStore. Прежде всего необходимо извлечь название хранилища и ключ (символ акции) из ассоциативного массива параметров запроса. Мы получаем объект HostInfo для запрашиваемого ключа и на основе содержащегося в нем имени хоста определяем, находится данный ключ в этом экземпляре или в каком-то удаленном.
- Далее мы проверяем, происходит ли (повторная) инициализация экземпляра Kafka Streams, на которую указывает возврат методом host() значения "unknown". Если да, то прекращаем обработку запроса и возвращаем сообщение "not accessible" («недоступен»).
- Наконец, мы проверяем, совпадает ли возвращенное имя хоста с именем хоста текущего экземпляра. Если нет, получаем данные с содержащего ключ экземпляра и возвращаем результаты.
- Я уже упоминал ранее, что мы скользим немного с запросом к оконному хранилищу, если параметры «от» и «до» в запросе отсутствуют. Если пользователь не указал промежуток времени, то по умолчанию мы вернем результаты за последнюю минуту из оконного хранилища. А поскольку длительность окна у нас равна 10 секундам, то мы вернем результаты за шесть окон. После извлечения сегментов окон из хранилища мы проходим по ним в цикле, формируем ответ, включающий число акций, приобретенных за каждый 10-секундный интервал времени в течение последней минуты

Listing 9.9 Mapping the request and checking for the key location

```

private String fetchFromWindowStore(Map<String, String> params) {
    String store = params.get(STORE_PARAM);
    String key = params.get(KEY_PARAM);
    String fromStr = params.get(FROM_PARAM);
    String toStr = params.get(TO_PARAM);
    HostInfo storeHostInfo = getHostInfo(store, key);
    if(storeHostInfo.host().equals("unknown")){
        return STORES_NOT_ACCESSIBLE;
    }
    if(dataNotLocal(storeHostInfo)){
        LOG.info("{} located in state store on another instance", key);
        return fetchRemote(storeHostInfo,"window", params);
    }
}

```

Extracts the request parameters

Gets the HostInfo for the key

If the hostname is "unknown", returns an appropriate message

Checks whether the returned hostname matches the host of this instance

Listing 9.10 Retrieving and formatting the results

```

Instant instant = Instant.now();
long now = instant.toEpochMilli();
long from = fromStr != null ? Long.parseLong(fromStr) : now - 60000;
long to = toStr != null ? Long.parseLong(toStr) : now;
List<Integer> results = new ArrayList<>();
ReadOnlyWindowStore<String, Integer> readOnlyWindowStore =
    kafkaStreams.store(store,
    QueryableStoreTypes.windowStore());
try(WindowStoreIterator<Integer> iterator =
    readOnlyWindowStore.fetch(key, from, to)){
    while (iterator.hasNext()){
        results.add(iterator.next().value());
    }
}
return gson.toJson(results);

```

Gets the current time in milliseconds

Sets the window segment start time or, if not provided, the time as of one minute ago

Sets the window segment ending time or, if not provided, the current time

Retrieves the ReadOnlyWindowStore

Fetches the window segments

Builds up the response

Converts the results to JSON and returns to the requestor

* событие это полное-состояние или дельта

2 января 2021 г. 9:25

событие это полное-состояние или дельта

<https://www.confluence.io/blog/messaging-single-source-truth/>

Одна тонкость заключается в том, как моделируются события. Они могут быть значениями: целыми фактами (Заказ в целом) или набором «дельт», которые необходимо повторно объединить (целое сообщение Заказ, за которым следуют сообщения обозначающие только изменения состояния: «сумма обновлена до \$ 5 », к заказу отмечен и т. д.).

В качестве аналогии представьте, что вы создаете систему контроля версий. Когда пользователь впервые фиксирует файл, вы его сохраняете. Последующие коммиты могут сохранять только «дельту»: только строки, которые были добавлены, изменены или удалены. Затем, когда пользователь выполняет проверку, вы открываете файл версии 0 и примените все дельты для перехода к текущему состоянию.

Альтернативный подход - просто сохранить весь файл в том виде, в котором он был на момент изменения. Это позволяет быстро и легко извлечь версию, но для сравнения разных версий вам придется выполнить «различие».

если я выбрали в событии передавать полное состояние, то я могу легко извлечь последнюю версию, чтобы всегда можно было вычислить дельту. [\[Более для извлечения разницы\]](#)

<https://martinfowler.com/eaDev/EventSourcing.html>

```
{  
  eventType: ProfileEditEvent,  
  timestamp: 1413215518,  
  profileId: 1234,  
  old: {  
    location: "San Francisco, CA",  
    industry: "Internet",  
  },  
  new: {  
    location: "New York, NY",  
    industry: "Financial Services"}  
}
```

Обратные события
Помимо событий, которые разыгрываются вперед, им также часто бывает полезно иметь возможность развернуться.

Реверсирование происходит наиболее просто, когда событие приводится в виде разности. Примером этого может быть «добавить 10 долларов на счет Мартина» вместо «постановить на счет Мартина 110 долларов». В первом случае я могу отменить, просто вычесть 10 долларов, но во втором случае у меня недостаточно информации для воссоздания прошлой стоимости учетной записи.

Если входные события не соответствуют разностному подходу, тогда событие должно гарантировать, что оно хранит все необходимое для разворота во время обработки. Вы можете сделать это, сохранив предыдущие значения для любого изменяемого значения или вычислив и сохранение различия в событии.

Это требование к хранению имеет значительные последствия, когда логика обработки находится внутри модели предметной области, поскольку модель предметной области может изменять свое внутреннее состояние способами, которые не должны быть видимыми для обработки объекта события. В этом случае всего разработать модель предметной области, чтобы знать о событиях и иметь возможность использовать их для хранения предыдущих значений.

Стоит помнить, что все возможности ображения событий могут быть реализованы вместо этого путем возврата к прошлому моментальному снимку и воспроизведения потока событий. В результате для функциональности никогда не требуется реверсирования. Однако это может иметь большое значение для эффективности, так как вы часто можете оказаться в положении, когда реверсирование нескольких событий намного эффективнее, чем использование игры вперед на большом количестве событий.

Эти два подхода в равной степени применимы к данным, которые мы храним в журнале. Итак, если взять более ориентированный на бизнес пример, заказ обычно представляет собой набор позиций (т. е. Вы часто заказываете несколько разных позиций за один покупку). При реализации системы, обрабатывающей покупки, вы можете задаться вопросом: должна ли заказ быть смоделирован как событие одного заказа со всеми позициями внутри него, или каждая позиция должна быть отдельным событием, а заказ перекомпонован путем сканирования различных индексов, незавершенные позиции? In domain-driven design, an order of this latter type is termed an aggregate (as it is an aggregate of line items) with the wrapping entity — that is, the order—being termed an aggregate root

Объекты событий, генерируемые производителем сенсорного типа, обычно записывают текущее значение того, что датчик измеряет. Производитель сенсорного типа может использовать три стратегии относительно того, когда он передает событие. Во-первых, событие должно генерироваться только тогда, когда оно обнаруживает заметное изменение значения того, что оно измеряет; например, датчик температуры может сгенерировать событие, если температура изменилась более чем на 0,5 ° С от последнего сообщенного значения. Во-вторых, блок кода всегда кодирует событие, потому что физически сообщает о состоянии или изменении состояния. В этом случае оно периодически сообщает текущее значение, каким бы оно ни было, и третий подход никогда не генерирует событие спонтанно, а вместо этого записывает текущее значение и возвращая его любому, кто его запросит. Этот третий подход иногда называют опросом и похож на распределение событий в стиле pull,

вариант1) первое событие полностью, дальнейшие события только дельта

- В качестве аналогии представьте, что вы создаете систему контроля версий. Когда пользователь впервые фиксирует файл, вы его сохраняете. Последующие коммиты могут сохранять только «дельту»: только строки, которые были добавлены, изменены или удалены. Затем, когда пользователь выполняет проверку, вы открываете файл версии 0 и примените все дельты для перехода к текущему состоянию.

вариант 2) каждое событие содержит файл полностью

- Альтернативный подход - просто сохранить весь файл в том виде, в котором он был на момент изменения. Это позволяет быстро и легко извлечь версию, но для сравнения разных версий вам придется выполнить «различие».
- Альтернативный подход - просто сохранить весь файл в том виде, в котором он был на момент изменения, для каждой отдельной фиксации. Очевидно, что для этого потребуется больше места для хранения, но это означает, что извлечение определенной версии из истории - это быстрое и легкое извлечение файла. Однако, если пользователь хочет сравнить разные версии, система должна использовать функцию «diff».

journal the whole fact as it arrived

правило: неизменно фиксируется именно то, что вы получаете [\[пришел полный файл\]](#), [\[фиксируем в виде события полный файл\]](#), [\[пришла дельта фиксируем в виде события дельта\]](#), [\[не лучше не складывать информацию о событии и не разбивать одно событие на несколько\]](#)

- т.е если заказ пришел со всеми товарами, то фиксируем в событии все
- а если пришла только отмена одного товара (и нет остальных позиций) то фиксируем в событии только один тов
- несколько практических правил, которые могут помочь. Самый важный из них - записывать все факты по мере их поступления. Поэтому, когда пользователь создает заказ, если этот заказ появляется со всеми позициями внутри него, мы обычно записываем его как единый объект.
- Но что происходит, когда пользователь отменяет одну позицию? Простое решение - просто новая запись все это в журнал как еще один агрегированный, но отмененный. Но что, если при какой-то причине заказ недоступен, а все, что мы получаем, - это одна отмененная позиция?
- Эмпирическое правило - записывать то, что вы получаете, поэтому, если поступает только одну позицию товара то записываем ее в журнал [\[Процесс обединения со всеми остальными товарами в заказе можно выполнить потом по стечению из материализован view\]](#).
- И наконец, разбиение событий на подсобытия по мере их появления часто не является хорошей практикой по тем же причинам.
- см пример "The keys used to partition the event streams must be invariant if ordering is to be guaranteed" https://v1systemdesign.onenoteit.algonote.v1Sync\cova_from.onenoteit\algonote.v1\key20hashing.onenote%20key§ion_id=1f14f3456005&page_id=6679A20-9F99-4B01-9EBD-88AC75635216&bend
- см также правило «для того чтобы все это заработало нужно собирать данные в реальном времени (как в реактивном программировании) https://v1systemdesign.onenoteit.algonote.v1Sync\cova_from.onenoteit\algonote.v1SYSTEMDESIGN\kafka1\EVENT%20DRIVEN.onenote#EVENT%20DRIVEN§ion_id=71728804-BDE4-4FFF-A011-59DE7979A941&page_id=1F7E4313-F766-4981-B2FA-C2E28E79181&bend

тогда для того чтобы все это заработало нужно собирать **данные в реальном времени** (как в **реактивном программировании**) тут сразу и мгновенно как только событие возможно поймать у источника в том объеме каком он возник

- Я думаю, что отсутствие сбора данных в реальном времени, вероятно, обрекло коммерческие системы потоковой обработки. Их клиенты по-прежнему занимались файловой, ежедневной пакетной обработкой для ETL и интеграции данных. Но оказалось, что в то время очень немногие люди действительно имели потоки данных в реальном времени
- Когда данные собираются пачками, это почти всегда происходит из-за некоторого ручного действия или отсутствия цифровых или является историческим реликтом, оставшимся от автоматизации какого-либо нецифрового процесса
- Рабочие «пакетные» задания на обработку, которые выполняются ежедневно, часто эффективно имитируют вид непрерывных вычислений с размером окна в один день.

в примере: один order содержит только один product (а не массив)

- Т.к. мы можем обработать только один order
- это находится в согласии с законом "journal the whole fact as it arrived см "onelogit #*%20событие%20это%20полное-состояние%20или%20делта§ion-id=[53A24205-3590-4EC8-BF8F-97057f636f61&end&base-path=C:\Users\trans\Sync\vova_from_onelogit\file\algonote_v1\SYSTEMDESIGN\KAFKA\1_EVENT\20DRIVEN_one"
- Но что происходит, когда пользователь отменяет одну позицию? Простое решение - просто снова записать все это в журнал как еще один агрегированный, но отмененный. Но что, если по какой-то причине заказ недоступен, а все, что мы получаем, - это одна отмененная позиция?
- Эмпирическое правило - записывать то, что вы получаете, поэтому, если постулат **записите только одну позицию товара** то записите только ее одну. Процесс объединения со всеми остальными товарами в заказе можно выполнить потом по членению из materialized view.
- И наоборот, разбиение событий на подсобытия по мере их появления часто не является хорошей практикой по тем же причинам.
- См также правило "для того чтобы все это заработало нужно собирать данные в реальном времени как в реактивном программировании" onelogit EVENT%20DRIVEN_one&EVENT%20DRIVEN§ion-id=[71728804-BE44-4FFF-A011-59DE9779A04]&page-id=1&P7e4313-f7e6-4981-b27a-c2e28eb79181&end&base-path=C:\Users\trans\Sync\vova_from_onelogit\file\algonote_v1\SYSTEMDESIGN\KAFKA

per-event processing

- Обработка событий (per-event processing) означает, что каждая запись обрабатывается сразу же, как только оказывается доступна, никакой группировки данных в небольшие пакеты (микропотокований) не нужно.
- Важно помнить следующее: потоковая обработка — оптимальный подход для случая, когда выдать уведомление или предпринять какие-либо действия необходимо сразу же при поступлении данных.
- Если же требуется углубленный анализ или сбор большого архива данных для дальнейшего анализа, возможно, потоковая обработка не то, что вам требуется

чувствительны ко времени в том смысле, что **действия необходимо предпринимать** при первой же возможности.

- В идеале сбор информации должен производиться при появлении событий.

Сам факт использования потокового приложения подразумевает ваше желание обрабатывать данные сразу же после их появления.

Более быстрая реакция.

- Автоматическая обработка непрерывных потоков событий в режиме, близком к реальному времени, позволяет бизнесу реагировать на эти события в течение нескольких минут или даже секунд.
- Непрерывный поток событий представляет собой «пульс» бизнеса и делает обычное хранение данных с пакетной загрузкой устаревшим по сравнению с ним.

event enrichment - источник включает в событие как можно больше информации

- на случай что она может кому-то понадобиться
- **источник потока данных** (см выше "тогда для того чтобы все это заработало нужно собирать данные в реальном времени, как в реактивном программировании") **не собирать появление события, как только оно произошло**

паттерн связан с "event carried state transfer"

- В качестве альтернативы можно использовать обогащение событий — это когда событие содержит информацию, которая нужна потребителю. В результате потребители событий становятся более простыми, поскольку **ниже не нужно запрашивать данные у сервиса, опубликовавшего событие**. Агрегат Order может обогатить событие OrderCreated, включив в него информацию о заказе

Обогащение событий упрощает потребителей

- Класс события OrderCreated, показанный ранее, отражает суть произошедшего. Но при обработке события OrderCreated вашему потребителю могут понадобиться подробности о заказе. Он может извлечь эту информацию из класса OrderService. Но недостатком запрашивания агрегата у сервиса являются накладные расходы на выполнение этого запроса.

из-за этого структура событий может часто меняться

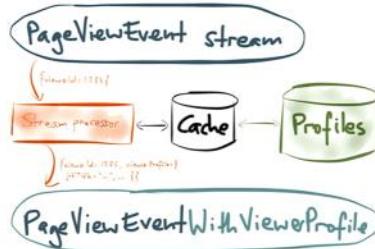
- снижения стабильности классов событий. Эти классы потенциально могут меняться каждый раз, когда обновляются требования их потребителей. Это способно отрицательно сказаться на поддерживаемом, поскольку изменения такого рода могут затронуть несколько частей приложения. К тому же удовлетворение требований каждого потребителя может оказаться недостижимой целью. К счастью, во многих ситуациях довольно легко определить, какие свойства следует включить в событие.

правильный event enrichment (по клепману)

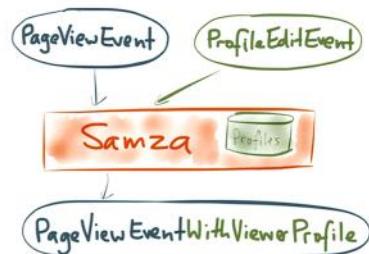
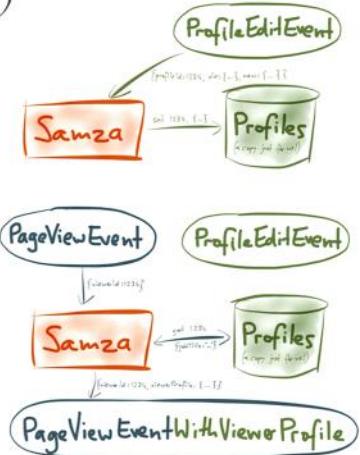
Базы 1)



Базы 2)



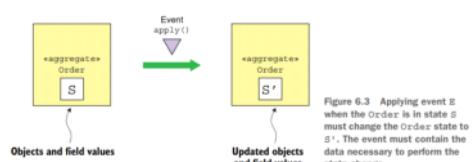
Базы 3)



Событие должно содержать данные, необходимые агрегату для перехода к новому состоянию.

см event - это изменение состояния агрегата `onEvent:///C:/Users/trans\Qvync\vova_from_onenote\if_agonote_v1\SYSTEMDESIGN\KAFA_EVENT%20DRIVEN_oneEvent%20ourcing§ion-id=2172BBD4-BDE4-4FFF-A011-59DE97794941&page-id=2E7242C2D-0850-485C-951F-7801AD8616E7!&end`

- Состояние агрегата включает в себя значения полей его объектов.
- Изменение состояния может заключаться в простом изменении поля объекта, такого как Order.state.
- Оно также может подразумевать добавление или удаление других объектов, таких как позиции заказа.
- Некоторые события, такие как Order Shipped, содержит немного данных (или не содержит никаких) и просто описывают переход состояния. Метод apply() обрабатывает событие Order Shipped, меняя поле заказа на SHIPPED.
- Но есть события, которые несут в себе много информации. Например, событие OrderCreated должно содержать все данные, необходимые методу apply() для инициализации заказа, включая его позиции, сведения о платеже, доставке и т. д.
- Suppose, as Figure 6.3 shows, that the current state of the aggregate is S and the new state is S'. An event E that represents the state change must contain the data such that when an Order is in state S, calling order.apply(E) will update the Order to state S'. In the next section you'll see that apply() is a method that performs the state change represented by an event.



в сага-хореографии иventы должны генерироваться агрегатом, даже если состояние не меняется

- Проблема использования событий для хореографии повествований состоит в их двойном назначении. В шаблоне «Порядок срабатывания» они описывают изменение состояния, но хореографии повествования должны генерироваться агрегатом, даже если состояние не меняется. Например, если обновление агрегата нарушает бизнес-правило, тот должен генерировать событие, чтобы сообщить об ошибке. Еще более серьезная проблема может возникнуть, когда участнику повествования удастся создать агрегат, в этом случае вернуть ошибку непросто некому

версия от Eberhard Wolf

какие данные передаются вместе с событием

(1) для каждого получателя свой тип события, только с нужной потребностью информации

- Если данные должны использоваться для таких разных целей, как написание счета, статистики или рекомендаций, то в событии должно храниться большое количество различных атрибутов.
 - Для выставления счетов, цены и налоговые ставки должны быть известны.
 - Для доставки необходим размер и вес товара, чтобы организовать подходящий транспорт.
- Решением может быть предоставление определенных типов событий для каждого получателя. Каждый тип события будет содержать информацию, которая нужна этому получателю.
 - Если в системе появляется новый заказ, в систему выставления счетов отправляется событие с необходимыми данными. Другое событие отправляется на отправку с данными для этой системы.
 - Эти две системы действительно отделены друг от друга: изменение интерфейса одной из систем не влияет на другую систему. Изменение информации в типе события для одной системы не повлияет на другую систему (так, как у нее свой тип события)

(2) Другое решение было использовать `public interface`

- В этом случае существует общая структура данных, которая содержит всю информацию для всех получателей. Это может затруднить понимание того, какие получатели используют, что, поскольку изменения в структуре данных могут привести к непредвиденным проблемам. Однако существует только одна структура данных, поэтому реализовать систему несколько проще.

- Очень важным вопросом является разница в информации, которая нужна каждому получателю. Если это в основном то же самое, опубликованный язык может быть лучше. Если он сильно отличается, возможно, имеет смысл иметь отдельные структуры данных.

(3) [соправительство языка идентификаторов \(команд\)](#), а данные можно подтянуть по идентификатору самостоительно

- После этого каждый ограниченный контекст может решить для себя, как получить необходимые данные. Для каждого ограниченного контекста может быть специальный интерфейс, который предоставляет соответствующие данные для этого конкретного ограниченного контекста

Однако реализовать этот подход не так просто. Ключевой вопрос - какие данные передаются с событием. Если данные будут использоваться для таких разных целей, как написание счета, статистики или рекомендаций, тогда в событии должно быть сохранено большое количество различных атрибутов

- Например, с [паттерн customer/supplier](#) группой по Shipping Microservice и Invoice Microservice может определяться, какие данные ей необходимо получить. Команда, предоставляющая данные для заказа, должна соответствовать этим требованиям. Этот шаблон определяет взаимодействие команд, которые разрабатывают ограниченные контексты, участвующие в коммуникационных отношениях. [Такая корпоративная необходимость](#) независима от того, отправляются ли события или происходит ли обмен данными между компонентами посредством синхронного вызова.

вариант 1 от Eberhard Wolf: разбить событие на несколько по одному на каждый зависимый микросервис (это антипаттерн так как лучше событие не разбивать см выше)

- Решением может быть предоставление определенных типов событий для каждого получателя. Каждый тип события будет содержать только ту информацию, которая нужна этому получателю. Таким образом, если в системе появляется новый заказ, в систему выставления счетов отправляется событие с необходимыми данными. Другое событие отправляется в службу доставки с данными для этой системы. Две системы действительно разделены: изменение интерфейса одной из систем не влияет на другую систему. Это может быть результатом взаимоотношений покупателя и поставщика.
- плюс в том что для каждого зависимого микросервиса свою схему сообщения (т-ф: хотя схема и не должна зависеть от зависимого микросервиса)

вариант 2 от Eberhard Wolf: DDD published language pattern

- Другое решение - использовать опубликованный язык (см. Раздел 2.1). В этом случае существует общая структура данных, которая содержит всю информацию для всех получателей. Это может затруднить понимание того, какие приемы используются для изменения данных. Но это делает возможным предсказывать будущее. Однако существует только одна структура данных, поэтому разрабатывать систему несколько проще.
- Очень важный момент - различия в информации, которая нужна каждому получателю. Если это в основном то же самое, опубликованный язык может быть лучше. Если он сильно отличается, возможно, имеет смысл иметь отдельные структуры данных

вариант 3 от Eberhard Wolf (антипаттерн к event carried state transfer): отправляется только идентификационный номер, например, номер нового заказа.

- Есть еще один способ справиться с этой проблемой. В этом случае отправляется только идентификационный номер, например, номер нового заказа. Впоследствии каждый ограниченный контекст может решить для себя, как получить необходимые данные. Для каждого ограниченного контекста может быть специальный интерфейс, который предоставляет соответствующие данные для этого конкретного ограниченного контекста.

пример 4 от Eberhard Wolf: в конце серии мелких событий публикуют [один обогащенный](#) событие (загружаясь в тему все меньше)

- <https://www.innog.com/en/blog/domain-events-versus-event-sourcing/>
- С моей точки зрения, этот подход нарушает предполагаемую инкапсуляцию между различными частями системы, приводит к болтовне между ограниченными контекстами, и таким образом, увеличивает связь между ограниченными контекстами. Основная причина заключается в том, что семантика детализированных событий из источника событий слишком низкоуровневая, как с точки зрения самого события, так и связанной с ним информации (только ID).
- На мой взгляд, если читать дальше, если UserRegistration#register будет также опубликовывать событие domen UserRegistrationCompleted со всей соответствующей информацией MobileNumber, FullName, Address#, например, не VerificationCode) в качестве полезной нагрузки после успешной регистрации завершена. Это событие предметной области имеет соответствующую семантику, чтобы его можно было легко обрабатывать внешними ограниченными контекстами без необходимости знать какие-либо внутренние механизмы процесса регистрации.

derived events

- Когда обнаруживается брошенная корзина, создается новое событие « Покупатель бросает корзину » в соответствии с предыдущим алгоритмом. Но куда мы должны писать это новое событие? У нас есть два варианта:
 - Мы записываем его обратно в ту же тему `raw-events-ch05` , совмещая его с исходными событиями, чтобы все события можно было вновь алгоритмически обработать.
 - Или в том что теперь все события можно найти в одном потоке; независимо от места, откуда они пришли.
- Мы пишем это в новую тему Kafka, которая называется производные-события-ch05
- запись в отдельный поток, производные-события-ch05 , дает понять, что эти новые события являются событиями второго порядка . производными от исходных событий.
- Инженер по обработке данных, который заботится только об этом событии (например, для отправки электронных писем о брошенной корзине покупок), может прочитать этот новый поток и игнорировать исходный поток.

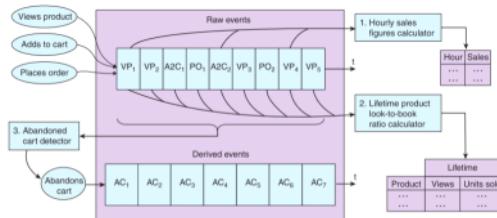


Figure 5.2 Our abandoned shopping cart detector consumes events from the `raw-events-ch05` topic in Kafka and generates new Shopper abandons cart events to write back to a new Kafka topic, called `derived-events-ch05`.

Data volatility

- At this point, you might be wondering why some of our data points are embedded in the event (such as the delivery truck's mileage), whereas other data points (such as the delivery truck's year of registration) are joined to the event later, in Redshift. You could say that the data points embedded inside the event were early, or eagerly joined, to the event, whereas the data points joined only in Redshift were late, or lazily joined.
- the answer comes down to the volatility, or changeability, of the individual data points. We can broadly divide data points into three levels of volatility:
 - Мы можем в общих чертах разделить точки данных на три уровня волатильности:
- The volatility of a given data point is not set in stone: a customer's surname might change after marriage, whereas the truck's mileage won't change while it is in the garage having its oil changed.
- But the expected volatility of a given data point gives us some guidance on how we should track it:
 - volatile data points should be eagerly joined in our event tracking, assuming they are available.
 - Unchanging data points can be lazily joined later in our unified log.
 - For slowly changing data points, we need to be pragmatic; we may choose to eagerly join some and lazily join others.



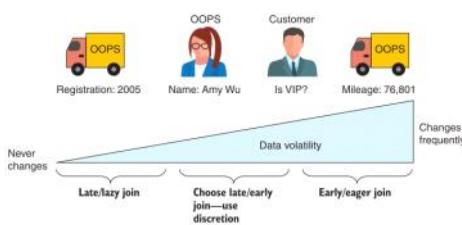


Figure 10.15 The volatility of a given data point influences whether we should attach that data point to our events "early" or "late."

пример здесь: [* analytics on event streams](#) [* event storming](#)

Stable data points

- For example, the delivery truck's year of registration or the delivery driver's date of birth

Slowly or infrequently changing data points

- For example, the customer's VIP status, the delivery driver's current job role, or the customer's name

Volatile data points

- For example, the delivery truck's current mileage

версия от мартина клепмана

событие отражает действия пользователя, а не детали обновления состояния, которое произошло в результате этого действия

- Событие CDC для обновления записи, как правило, содержит всю новую версию записи, вследствие чего текущее значение для первичного ключа полностью определяется самим последним событием этого ключа. При уплотнении журнала предыдущие события для данного ключа могут быть отброшены.
- Применимо к источникам событий, наоборот, события моделируются на более высоком уровне: обычно событие отражает действия пользователя, а не детали обновления состояния, которое произошло в результате этого действия. В таком случае последующие события обычно не перезаписывают предшествующие, так что для восстановления конечного состояния нужна полная история событий. Уплотнение журнала в данном случае невозможно

Когда поступает запрос от пользователя, вначале он представляет собой команду

- На данном этапе тоже вероятен сбой — например, при нарушении некоего условия целостности

Сначала приложение должно проверить, может ли выполнить команду

- Например, если пользователь пытается зарегистрироваться под каким-либо именем, или забронировать место в самолете, или купить билет в театр, то приложение должно убедиться, что это имя или место еще не занято
- При успешном прохождении проверки приложение может генерировать событие для демонстрации того, что данное имя пользователя зарегистрировано и ему присвоен определенный ID или что место зарезервировано для данного клиента.

Если проверка прошла успешно и команда принятия, то она становится событием, которое является долговечным и неизменным

- В этот момент, когда событие генерировано, оно становится фактом

Даже если клиент впоследствии изменит свое решение или отменит бронирование, факт остается фактом: ранее он сделал забронировку определенного места, а изменение или аннулирование — это уже другое событие, которое добавлено позже.

Потребители потока событий не разрешают откладывать события к тому времени, когда потребитель увидит событие, оно уже является неизменной частью журнала. Возможно, это уже видели другие потребители. Таким образом, любая проверка команды должна выполняться синхронно, прежде чем она станет событием, — например, с помощью сериализируемой транзакции, которая проверяет команду как неделимую единицу и затем публикует событие

- Вероятен другой вариант запроса пользователя на бронирование места может быть разделен на два события — сначала предварительное резервирование, а затем отдельное событие подтверждения после проверки бронирования (как было описано в пункте «Реализация линеаризуемого хранилища с помощью рассылки общей последовательности» подраздела «Рассылка общей последовательности» раздела 9.3). Такое разделение позволяет выполнять валидацию в асинхронном процессе.

разное

событие имеет значение вне контекста конкретного взаимодействия между его производителем и потребителями.

- События имеют значение, которое не зависит от его производителя и его потребителей, и в результате производители событий и потребители событий могут быть полностью отделены друг от друга. An event has meaning that is independent of its producer and of its consumers, and as a result event producers and event consumers can be completely decoupled from each other.
- Идея использования самого события для разделения производителя событий и потребителя событий является существенным различием между программированием, основанным на событиях, и дизайном приложений, основанным на взаимодействиях запроса и ответа, he idea of using the event itself to decouple the event producer and event consumer is a significant difference between event-based programming and application design based on request-response interactions.

данко коньюмер зависит от продьюсер (несмотря на то что продьюсер не зависит от коньюмера)

- Когда вы реализуете поставщика услуг, вы обычно кодируете его так, чтобы он предоставляет эту услугу другому коньюмеру, который делает запрос к запрашивающей услуге, поэтому поставщик отделен от запрашивающей стороны. Однако запрашивающая услуга зависит от поставщика услуг, выполняющего согласованную функцию.

стандарт на атрибуты события

атрибуты события

- некоторые атрибуты являются общими для всех типов событий, и поэтому они имеют структуру заголовка и полезной нагрузки

Table 3.2 The header attribute indicators for the Delivery Bid event type

Attribute type	Occurrence
occurrenceTime	required
eventAnnotation	optional
eventCertainty	not applicable
eventIdentity	required
detectedTime	required
eventSource	required

OASIS WsDM Event Format (стандарт описывающий заголовок события)

- предназначен главным образом для сети и систем управления приложений. Этот формат имеет богатую структуру заголовков, включая атрибуты категоризации, которые позволяют фильтровать и маршрутизировать события процессорами, которые не обязательно понимают весь тип события.

WS-Topics (стандарт описывающий payload события)

- is an OASIS standard that allows event types to be classified into topics. It assumes that an event payload is described using an XML Schema global element definition (as shown in listing 3.2).
- Эти две спецификации фокусируются на полезной нагрузке событий; их предположение состоит в том, что атрибуты, которые мы описываем как заголовки, либо включены в эти полезные данные, либо транспортируются вместе с полезными данными в другом месте сообщения веб-службы

WS-EventDescriptions

- is a specification developed by the W3C, to provide a standard way of describing event types used in web services. It also uses the approach shown in listing 3.2

Чтение тоже событие

Запросы чтения тоже можно представить в качестве потоков событий

- и пропускать через потоковый процессор как события записи, так и события чтения; в ответ на запросы чтения событий процессор будет помещать результат чтения в выходной поток

запись событий чтения в долговременное хранилище позволяет лучше отслеживать причинно-следственные зависимости

- Еще одно потенциальное преимущество от записи событий чтения в журнал — возможность отслеживать причинно-следственные зависимости и происхождение данных в системе. Это позволяет узнать, что увидел пользователь перед тем, как принял то или иное решение. Например, в интернет-магазине, вполне вероятно, предложенная дата доставки и наличие товара на складе, показанные клиенту, повлияют на выбор им товара для покупки [4]. Для анализа этой взаимосвязи необходимо записать результат пользовательского запроса об условиях доставки и наличие товара.

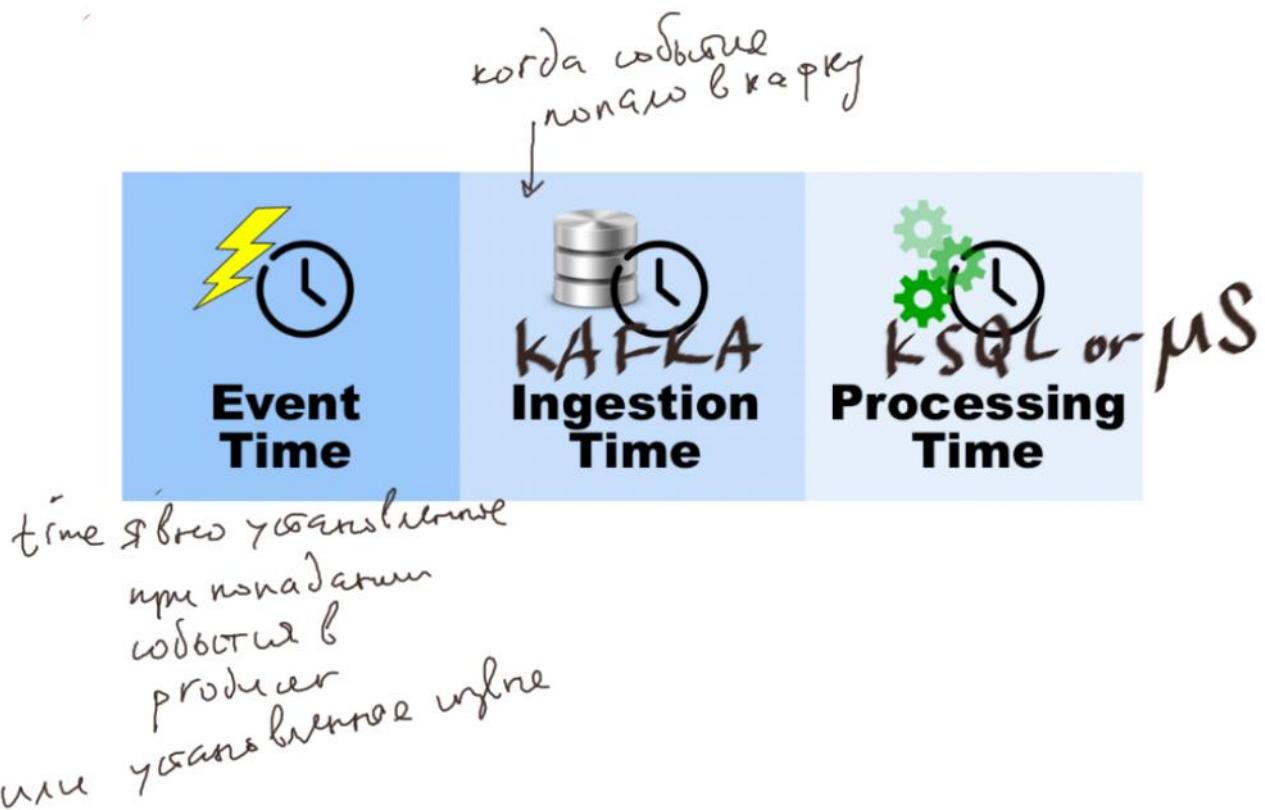
set event timestamp in producer

22 декабря 2020 г. 18:29

event time - как явно в проге-продьюсере присвоить таймстамп реального возникновения события, ТК он может отличаться от времени когда его забрала кафка

- далее даже если они перемешаются в кафке, их можно будет отсортировать через windowing

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



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

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

- Если не задать метку даты/времени, это сделает генератор (с использованием текущего системного времени) перед отправкой записи брокеру Kafka. На метки даты/времени также влияет параметр конфигурации брокера log.message.timestamp.type, который может принимать значение или CreateTime (по умолчанию), или LogAppendTime. Подобно многим другим настройкам брокеров, задаваемое для брокера значение применяется ко всем

топикам по умолчанию, но при создании топика можно задать для него другое значение. Если указать значение `LogAppendTime` и топик не перекрывает настроек брокера, то брокер будет записывать вместо метки даты/времени текущее время при добавлении записи в журнал. В противном случае будет использоваться метка даты/времени из объекта `ProducerRecord`.

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

processing time - Это разумно только в случае пренебрежительно малой задержки между созданием и обработкой событий

- Во многих потоковых системах обработки, напротив, для определения временного окна используются локальные системные часы вычислительной машины (время обработки) [79].
 - Преимущество такого подхода заключается в его простоте
 - Это разумно в случае пренебрежительно малой задержки между созданием и обработкой событий.
 - Однако при сколько-нибудь значительном отставании обработки, то есть если обработка может произойти заметно позже времени возникновения события, все нарушается

пример

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

Listing 4.13 Using the join() method

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

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

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

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

    Extracts the branched streams
    ValueJoiner instance used to perform the join
    Calls the join method, triggering automatic repartitioning of coffeeStream and electronicsStream
    Constructs the join
    Prints the join results to the console

```

Метки даты/времени играют важную роль в следующих ключевых аспектах функциональности Kafka Streams:

- Joining streams
 - Updating a changelog (KTable API)
 - Deciding when the Processor.punctuate() method is triggered (Processor API)

При использовании меток даты/времени безопаснее будет применять всемирное координированное время (UTC),

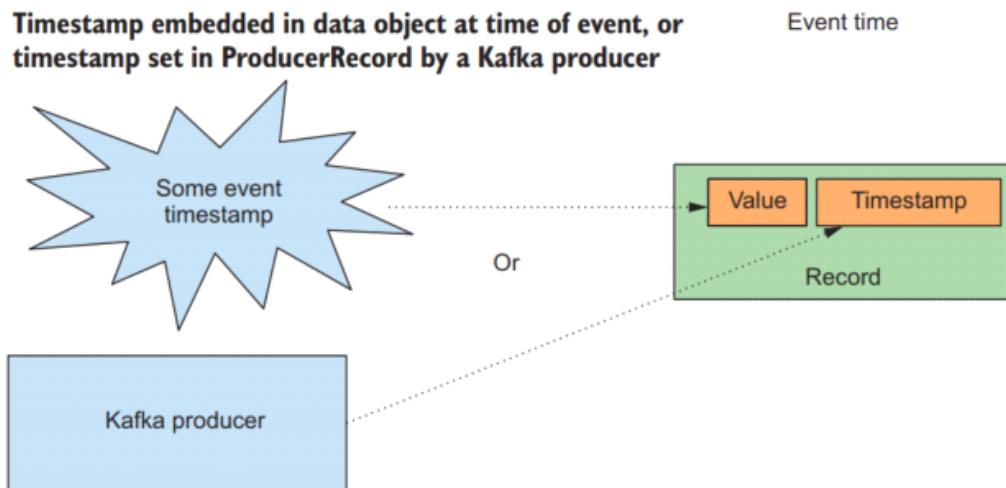
- До сих пор мы неявно предполагали, что клиенты и брокеры располагаются в одном часовом поясе, но это может быть не так. При использовании меток даты/времени безопаснее будет применять всемирное координированное время (UTC), чтобы исключить возможные недоразумения при использовании различными брокерами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

Метки даты/времени управляют движением данных в Kafka Streams. Следует тщательно и обдуманно выбирать источники меток даты/времени

можно сгруппировать метки даты/времени по трем категориям

(a) Время события (event time)

- метка даты/времени, задаваемая в момент генерации события, обычно включается в объект, представляющий событие.
 - A timestamp embedded in the actual event or message object (event-time semantics)
 - вы отправляете в Kafka сообщения с событиями, с зафиксированными в объектах сообщений метками даты/времени. Эти объекты событий становятся доступными генератору Kafka с некоторым запозданием, так что необходимо учитывать только включенную в объект метку даты/времени
 - требуется различать время потребления записей приложением Kafka Streams и время, указанное в метках даты/времени записей.
 - Для работы с включенными в значения записей метками даты/времени необходимо создать пользовательскую реализацию интерфейса TimestampExtractor см также ниже пример своего экстрактора
- Здесь мы будем считать временем события также и метку даты/времени, задаваемую при создании объекта ProducerRecord.
 - o Using the timestamp set in the record metadata when creating the ProducerRecord (event-time semantics)



(b) Время ввода данных (ingestion time)

- метка даты/времени, задаваемая в момент первого попадания данных в конвейер их обработки. Можно считать временем ввода данных метку даты/времени, устанавливаемую брокером Kafka (при параметре конфигурации log.message.timestamp.type, равном LogAppendTime).

Timestamp set at time record is appended to log (topic)

Ingest time

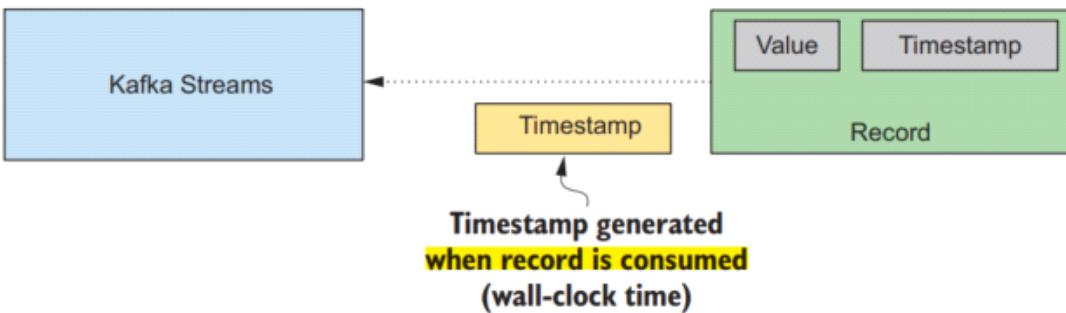


(c) Время обработки (processing time)

- метка даты/времени, задаваемая в момент начала прохождения данных или записи о событии через конвейер обработки.
- Using the current timestamp (current local time) when the Kafka Streams application ingests the record (processing-time semantics)

Timestamp generated at the moment when record is consumed, ignoring timestamp embedded in data object and ConsumerRecord

Processing time



пример готовых экстракторов

4.5.1 Provided TimestampExtractor implementations

Almost all of the provided `TimestampExtractor` implementations work with timestamps set by the producer or broker in the message metadata, thus providing either event-time processing semantics (timestamp set by the producer) or log-append-time

processing semantics (timestamp set by the broker). Figure 4.19 demonstrates pulling the timestamp from the `ConsumerRecord` object.

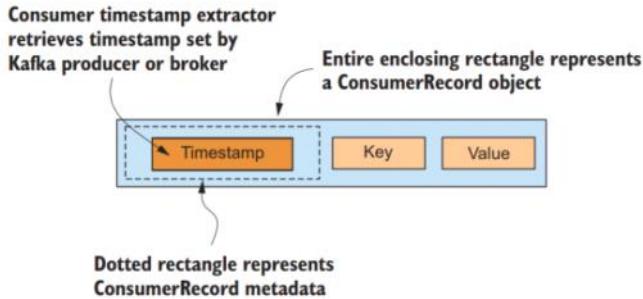


Figure 4.19 Timestamps in the `ConsumerRecord` object: either the producer or broker set this timestamp, depending on your configuration.

Although you're assuming the default configuration setting of `CreateTime` for the timestamp, bear in mind that if you were to use `LogAppendTime`, this would return the timestamp value for when the Kafka broker appended the record to the log. `ExtractRecordMetadataTimestamp` is an abstract class that provides the core functionality for extracting the metadata timestamp from the `ConsumerRecord`. Most of the concrete implementations extend this class. Implementors override the abstract method, `ExtractRecordMetadataTimestamp.onInvalidTimestamp`, to handle invalid timestamps (when the timestamp is less than 0).

Here's a list of classes that extend the `ExtractRecordMetadataTimestamp` class:

- `FailOnInvalidTimestamp`—Throws an exception in the case of an invalid timestamp.
- `LogAndSkipOnInvalidTimestamp`—Returns the invalid timestamp and logs a warning message that the record will be discarded due to the invalid timestamp.
- `UsePreviousTimeOnInvalidTimestamp`—In the case of an invalid timestamp, the last valid extracted timestamp is returned.

We've covered the event-time timestamp extractors, but there's one more provided timestamp extractor to cover.

4.5.2 `WallclockTimestampExtractor`

`WallclockTimestampExtractor` provides process-time semantics and doesn't extract any timestamps. Instead, it returns the time in milliseconds by calling the `System.currentTimeMillis()` method.

That's it for the provided timestamp extractors. Next, we'll look at how you can create a custom version.

пример своего экстрактора

- Пользовательский TimestampExtractor возвращает метку даты/времени на основе значения, содержащегося в объекте ConsumerRecord. Эта метка даты/времени может быть уже существующим значением или значением, вычисленным на основе свойств объекта-значения
- В примере с соединением мы применили пользовательский TimestampExtractor потому, что хотели задействовать метки даты/времени фактического момента покупки. Подобный подход дает возможность соединять записи даже в случае задержек доставки или поступления данных в неправильном порядке
- Старайтесь не слишком «умничать» при создании пользовательской реализации TimestampExtractor. Сохранение и архивирование журналов основываются на метках даты/времени, так что возвращаемая средством извлечения метка может стать меткой сообщения, которая затем будет применяться в журналах изменений и топиках результатов далее по конвейеру.

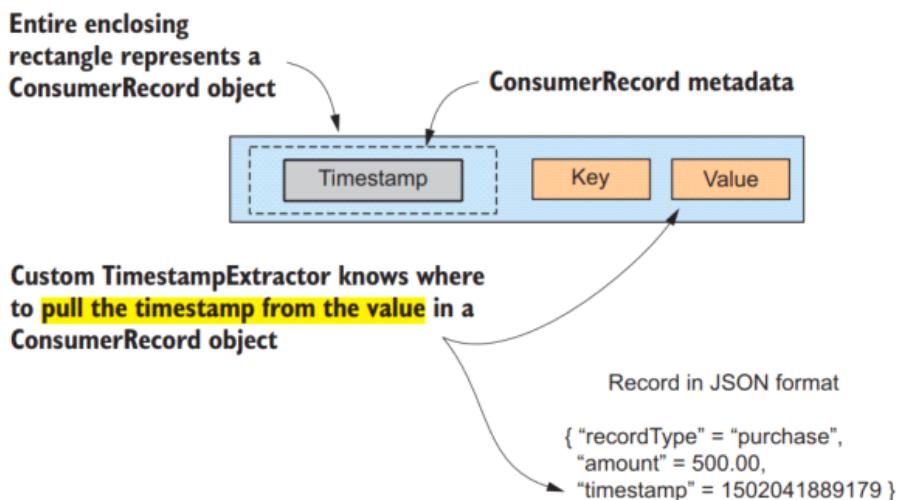


Figure 4.20 A custom TimestampExtractor provides a timestamp based on the value contained in the ConsumerRecord. This timestamp could be an existing value or one calculated from properties contained in the value object.

Listing 4.14 Custom TimestampExtractor

```
Retrieves the Purchase object from  
the key/value pair sent to Kafka
```

```
public class TransactionTimestampExtractor implements TimestampExtractor {  
  
    @Override  
    public long extract(ConsumerRecord<Object, Object> record,  
    long previousTimestamp) {  
        Purchase purchaseTransaction = (Purchase) record.value();  
        return purchaseTransaction.getPurchaseDate().getTime();  
    }  
}
```

Returns the timestamp
recorded at the point of sale

как указать, какой TimestampExtractor использовать

(a) задать глобальное средство извлечения меток,

- указав его в свойствах при настройке приложения Kafka Streams.
- Если же значение свойства не задано, будет применяться значение по умолчанию — FailOnInvalidTimestamp.class.
- Например, следующий фрагмент кода указывает приложению с помощью задания свойства при его настройке использовать класс TransactionTimestampExtractor:

```
props.put(StreamsConfig.DEFAULT_TIMESTAMP_EXTRACTOR_CLASS_CONFIG,  
        TransactionTimestampExtractor.class);
```

(b) передать экземпляр TimestampExtractor через объект Consumed

- Преимущество второго способа состоит в возможности использования своего TimestampExtractor для каждого источника входных данных, в то время как при первом варианте приходится обрабатывать записи из различных топиков в одном экземпляре TimestampExtractor.

```
Consumed.with(Serdes.String(), purchaseSerde)  
        .withTimestampExtractor(new TransactionTimestampExtractor()))
```

вы никогда не можете быть уверены в получении всех событий для определенного временного окна

- Например, предположим, что мы группируем события по однominутным интервалам с целью получить количество запросов в минуту. Мы подсчитали некоторое количество событий с отметками времени, соответствующими 37-й минуте часа. Но время идет дальше, и теперь большинство входящих событий приходится на 38-ю и 39-ю минуты часа. Когда мы сможем объявить об окончании обработки окна 37-й минуты и вывести его значение счетчика?
- Можно посчитать, что время истекло, и объявить временной интервал законченным после того, как в течение какого-то времени не поступало новых событий. Но может случиться так, что некоторые события задержались из-за сбоя сети и попали в буфер на другой машине. Нужно иметь возможность обрабатывать такие отставшие события, появившиеся после объявления временного интервала завершенным. В целом у нас есть два варианта [1].
 - o 1. Игнорировать отставшие события, поскольку их доля, скорее всего, в обычных условиях невелика. Можно отслеживать количество отброшенных событий как показатель и выдавать предупреждения, если система начнет отбрасывать значительный объем данных.
 - o 2. Опубликовать поправку, обновленное значение для данного временного окна с учетом отставших событий. В этом случае также может потребоваться отменить предыдущий вывод.
- Иногда можно использовать специальные сообщения вида: «С этого момента сообщения с временной меткой раньше чем t не будут поступать». Потребители могут применять такие сообщения для запуска временных окон [81].
- Однако если события генерируют несколько инициаторов на разных машинах, каждый из которых имеет собственные минимальные пороговые значения времени, то потребители должны отслеживать каждого производителя отдельно. Это вызывает сложности при добавлении и удалении инициаторов.

log three timestamps подход для определения event time

- Приложение может работать, пока мобильное устройство отключено от Сети. В этом случае оно будет буферизовать события локально и отправит их на сервер, когда появится подключение к Интернету (вероятно, через несколько часов или даже дней). Для всех

потребителей данного потока события будут появляться с большим отставанием

показания часов на устройстве, управляемом пользователем, часто не считаются достоверными, так как на них случайно или намеренно может быть установлено неправильное время

Время поступления события на сервер (по часам сервера), скорее всего, будет точным, поскольку он находится под вашим контролем, но менее значимым с точки зрения описания действий пользователя

Чтобы задействовать недостоверные показания часов устройства, применяется следующий подход с регистрацией трех временных меток

- 1) время, когда произошло событие, по часам устройства;
- 2) время, когда событие было отправлено на сервер, по часам устройства;
- 3) время, когда событие было получено сервером, по часам сервера

- Вычитая вторую временную метку из третьей, можно оценить смещение между часами устройства и сервера (при условии, что сетевая задержка незначительна по сравнению с требуемой точностью временной метки).
- Затем можно применить это смещение к временной метке события и таким образом оценить истинное время, в которое произошло событие (если между временем события и временем его отправки на сервер смещение часов устройства не изменилось)

* Punctuator

1 февраля 2021 г. 17:43

PUNCTUATION SEMANTICS (stream time)

- Главное в этом процессе то, что приложение наращивает метки даты/времени с помощью TimestampExtractor, так что вызовы метода punctuate() будут единообразными только в том случае, если данные поступают с постоянной скоростью. Если же данные поступают от случая к случаю, то метод punctuate() не будет выполняться через запланированные регулярные интервалы времени.
- Если вам нужно регулярно выполнять какие-либо действия независимо от движения данных, то оптимальным вариантом будет применение системного времени
- если требуется только производить вычисления над входящими данными и небольшой разрыв во времени между выполнениями допустим, попробуйте семантику потокового времени

Let's start our conversation on punctuation semantics with `STREAM_TIME`, because it requires a little more explanation. Figure 6.6 illustrates the concept of stream time. Let's walk through some details to gain a deeper understanding of how the schedule is determined (note that some of the Kafka Stream internals are not shown):

- 1 The StreamTask extracts the *smallest* timestamp from the PartitionGroup. The PartitionGroup is a set of partitions for a given StreamThread, and it contains all timestamp information for all partitions in the group.
- 2 During the processing of records, the StreamThread iterates over its StreamTask object, and each task will end up calling punctuate for each of its processors that are eligible for punctuation. Recall that you collect a minimum of 20 trades before you examine an individual stock's performance.
- 3 If the timestamp from the last execution of punctuate (plus the scheduled time) is less than or equal to the extracted timestamp from the PartitionGroup, then Kafka Streams calls that processor's punctuate() method.

The key point here is that the application advances timestamps via the TimestampExtractor, so punctuate() calls are consistent only if data arrives at a constant rate. If your flow of data is sporadic, the punctuate() method won't get executed at the regularly scheduled intervals.

With `PunctuationType.WALL_CLOCK_TIME`, on the other hand, the execution of `Punctuator.punctuate` is more predictable, as it uses wall-clock time. Note that system-time semantics is best effort—wall-clock time is advanced in the polling interval, and the granularity is dependent on how long it takes to complete a polling cycle. So,

In the two partitions below, the letter represents the record, and the number is the timestamp. For this example, we'll assume that `punctuate` is scheduled to run every five seconds.



Because partition A has the smallest timestamp, it's chosen first:

- 1) process called with record A
- 2) process called with record B

Now partition B has the smallest timestamp:

- 3) process called with record C
- 4) process called with record D

Switch back to partition A, which has the smallest timestamp again:

- 5) process called with record E
- 6) punctuate called because time elapsed from timestamps is 5 seconds
- 7) process called with record F

Finally, switch back to partition B:

- 8) process called with record G
- 9) punctuate called again as 5 more seconds have elapsed, according to the timestamps

Figure 6.6 Punctuation scheduling using STREAM_TIME

with the example in listing 6.6, you can expect the punctuation activity to be executed closer to every 10 seconds, regardless of data activity.

Which approach you choose to use is entirely dependent on your needs. If you need some activity performed on a regular basis, regardless of data flow, using system time is probably the best bet. On the other hand, if you only need calculations performed on incoming data, and some lag time between executions is acceptable, try stream-time semantics.

NOTE Before Kafka 0.11.0, punctuation involved the `ProcessorContext.schedule(long time)` method, which in turn called the `Processor.punctuate` method at the scheduled interval. This approach only worked on stream-time semantics, and both methods are now deprecated. I mention deprecated methods in this book, but I only use the latest punctuation methods in the examples.

Now that we've covered scheduling and punctuation, let's move on to handling incoming records.

6.3.3 The punctuator execution

We've already discussed the punctuation semantics and scheduling, so let's jump straight into the code for the `Punctuator.punctuate` method (found in `src/main/java/bejeck/chapter_6/processor/punctuator/StockPerformancePunctuator.java`).

Listing 6.8 Punctuation code

```
@Override
public void punctuate(long timestamp) {
    KeyValueIterator<String, StockPerformance> performanceIterator =
    ↪ keyValueStore.all();                                ←
    while (performanceIterator.hasNext()) {
        KeyValue<String, StockPerformance> keyValue =
    ↪ performanceIterator.next();
        String key = keyValue.key;
        StockPerformance stockPerformance = keyValue.value;

        if (stockPerformance != null) {
            if (stockPerformance.priceDifferential()
    ↪ >= differentialThreshold ||
```

Retrieves the iterator to go over all key values in the state store

Checks the threshold for the current stock

Listing 6.8 Punctuation code

```
@Override  
public void punctuate(long timestamp) {  
    KeyValueIterator<String, StockPerformance> performanceIterator =  
    → keyValueStore.all(); ←  
  
    while (performanceIterator.hasNext()) {  
        KeyValue<String, StockPerformance> keyValue =  
        → performanceIterator.next();  
        String key = keyValue.key;  
        StockPerformance stockPerformance = keyValue.value;  
  
        if (stockPerformance != null) {  
            if (stockPerformance.priceDifferential()  
            → >= differentialThreshold ||  
                stockPerformance.volumeDifferential()  
            → >= differentialThreshold) {  
                context.forward(key, stockPerformance); ←  
            }  
        }  
    }  
}
```

Retrieves the iterator to go over all key values in the state store

Checks the threshold for the current stock

If you've met or exceeded the threshold, forwards the record

The procedure in the `Punctuator.punctuate` method is simple. You iterate over the key/value pairs in the state store, and if the value has crossed over the predefined threshold, you forward the record downstream.

An important concept to remember here is that, whereas before you relied on a combination of committing or cache flushing to forward records, now you define the terms for when records get forwarded. Additionally, even though you expect to execute this code every 10 seconds, that doesn't guarantee you'll emit records. They must meet the differential threshold. Also note that the `Processor.process` and `Punctuator.punctuate` methods aren't called concurrently.

NOTE Although we're demonstrating access to a state store, it's a good time to review Kafka Streams' architecture and go over a few key points. Each `StreamTask` has its *own* copy of a *local* state store, and `StreamThread` objects don't share tasks or data. As records make their way through the topology, each node is visited in a depth-first manner, meaning there's never concurrent access to state stores from any given processor.

This example has given you an excellent introduction to writing a custom processor, but you can take writing custom processors a bit further by adding a new data structure and an entirely new way of aggregating data that doesn't currently exist in the API. With this in mind, we'll move on to adding a co-group processor.

Дескрипторы методов для класса Punctuator

Для экземпляра класса `Punctuator` можно задать дескриптор метода. Для этого нужно объявить метод в узле-обработчике, принимающий один параметр типа `long`, с возвращаемым типом `void`. После этого запланируйте пунктуацию следующим образом:

```
context().schedule(15000L, STREAM_TIME, this::myPunctuationMethod);
```

Пример этого вы можете найти в файле `src/main/java/bbejeck/chapter_6/processor/cogrouping/CogroupingMethodHandleProcessor.java`.

NOTE You'll recall from our previous discussion of punctuate semantics that you have two choices: `PunctuationType.STREAM_TIME` and `PunctuationType.WALL_CLOCK_TIME`. Listing 6.13 uses `STREAM_TIME` semantics. There's an additional processor example showing `WALL_CLOCK_TIME` semantics in `src/main/java/bbejeck/chapter_6/processor/cogrouping/CogroupingSystemTimeProcessor.java`, so you can observe the differences in performance and behavior.

Listing 6.15 The CogroupingPunctuator.punctuate() method

```
// leaving out class declaration and constructor for clarity
@Override
public void punctuate(long timestamp) {
    KeyValueIterator<String, Tuple<List<ClickEvent>, List<StockTransaction>>> iterator = tupleStore.all();           ← Gets iterator of all co-groupings in the store

    while (iterator.hasNext()) {
        KeyValue<String, Tuple<List<ClickEvent>, List<StockTransaction>>> cogrouping = iterator.next();           ← Retrieves the next co-grouping

        // if either list contains values forward results
        if (cogrouping.value != null &&
            (!cogrouping.value._1.isEmpty() ||           ← Ensures that the value is not null, and
            !cogrouping.value._2.isEmpty())) {           ← that either collection contains data

            List<ClickEvent> clickEvents =           ← Makes defensive
            new ArrayList<>(cogrouping.value._1);       ← copies of co-grouped
            List<StockTransaction> stockTransactions =   ← collections
            new ArrayList<>(cogrouping.value._2);
```

The co-group processor

167

```
        context.forward(cogrouping.key,
        Tuple.of(clickEvents, stockTransactions));           ← Forwards the key and aggregated co-grouping
        cogrouped.value._1.clear();
        cogrouped.value._2.clear();
        tupleStore.put(cogrouped.key, cogrouped.value);      ← Puts the cleared-out tuple back into the store
    }
}
iterator.close();
}
```

eventually consistent

19 декабря 2020 г. 23:20

Термин «согласованность в конечном счете» был придуман Дугласом Терри (Douglas Terry) и др. [24], популяризован известным Вернером Вогельсом (Werner Vogels) [22]

- и стал лозунгом множества MySQL-проектов. Однако не только базы данных NoSQL согласованы в конечном счете: ведомые узлы в асинхронно реплицируемых реляционных БД обладают теми же свойствами.

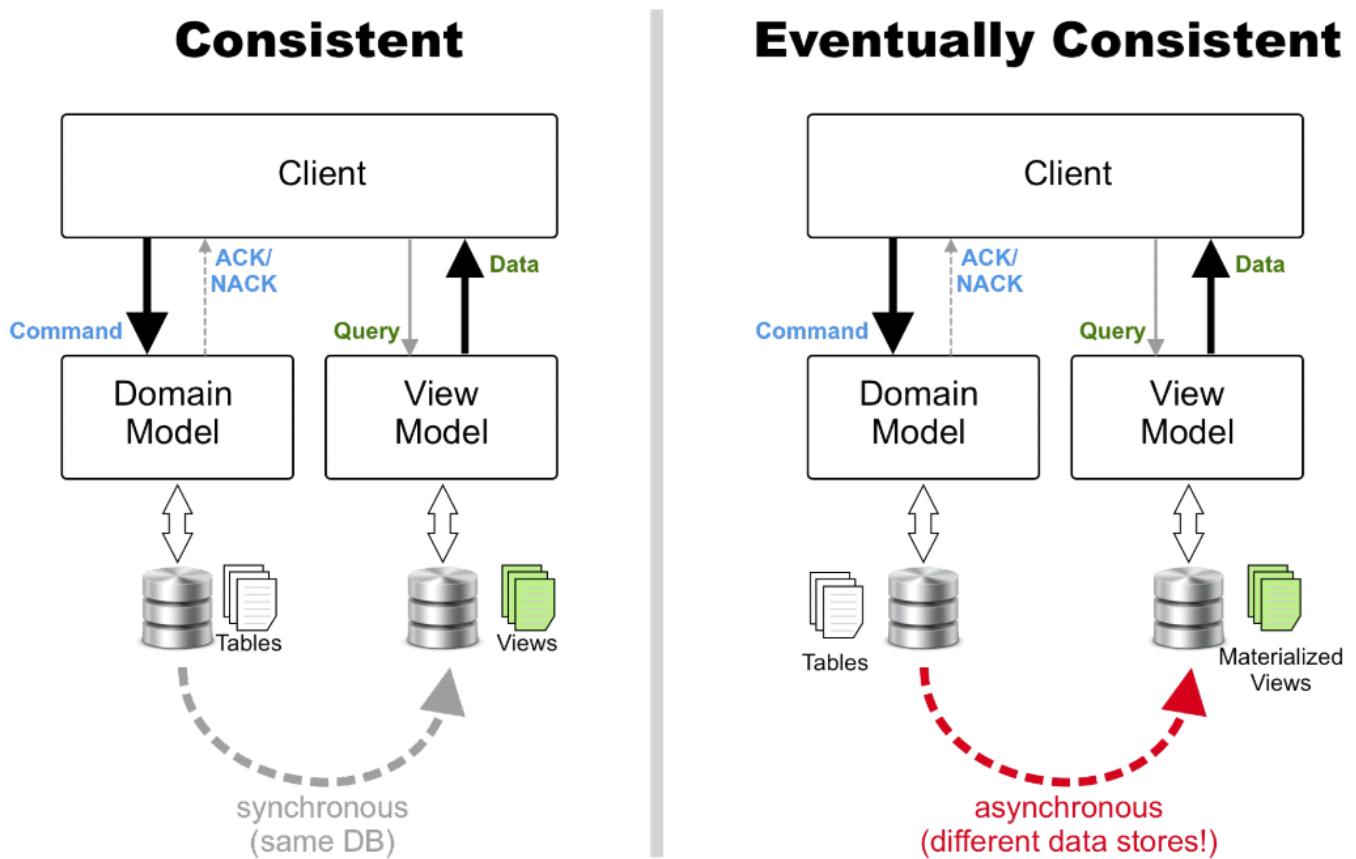
The word *consistency* is terribly overloaded:

- In [Chapter 5](#) we discussed *replica consistency* and the issue of *eventual consistency* that arises in asynchronously replicated systems (see “[Problems with Replication Lag](#)” on page 161).
- *Consistent hashing* is an approach to partitioning that some systems use for rebalancing (see “[Consistent Hashing](#)” on page 204).
- In the CAP theorem (see [Chapter 9](#)), the word *consistency* is used to mean *linearizability* (see “[Linearizability](#)” on page 324).
- In the context of ACID, *consistency* refers to an application-specific notion of the database being in a “good state.”

eventually consistent - означает что view model обновляется чуть с опозданием после domain model

- хотя это противоречит паттерну обновления даже доменной БД через кафку
- немного отличается от идеи [event carried state transfer](#) который как бы говорит не о том что "потом будет консистентно" а наоборот "уже заранее консистентно"

Consistent vs. Eventually-Consistent



The model of CQRS that we saw on the previous slide does not scale well because everything has to be consistent and thus synchronous. The next step was to introduce **eventual consistency** into the picture.

- On the left side we have the fully consistent view where the query that is executed right after an update caused by a command provides the new state
- This is often realized by having a single transactional DB as the target of a change triggered by a command and the source of data requested by a query
- Table changes triggered by the command are reflected immediately in the database views that are providing the data to the queries
- Now we can decouple the data store for the state of the domain model and the data store for the view model as shown on the right side of the image on the slide
- The source of the data requested by the queries in this slightly changed architectural pattern are often called **materialized views** (instead of only **views**)

eventually consistent - означает, что изменениям данных требуется время для перемещения между распределенными репликами, и в течение этого времени внешние наблюдатели могут видеть несогласованные данные

- Квалификация «eventually consistent» означает временное окно, в течение которого может наблюдаться несогласованность после ограничения изменения; когда система больше не получает модификаций и переходит в состояние покоя, она в конечном итоге снова становится полностью согласованной.

как только масштаб системы вырастает до критического размера, она уже не может быть **strongly consistent**

- Стоимость координации единого глобального порядка всех изменений, которые происходят в нем, будет недопустимо высокой, а добавление дополнительных (распределенных) ресурсов в этот момент только уменьшит пропускную способность системы, а не увеличит ее. Вместо этого мы будем строить системы из небольших строительных блоков - сущностей З - которые внутренне согласованы, но взаимодействуют eventually consistent

event sourcing

21 декабря 2020 г. 23:56

event sourcing - подразумевает что сообщения хранятся в кафке вечно (в качестве single source of truth)

- те никогда не удаляются из кафки
- One of the bigger differences between Kafka and other messaging systems is that it can be used as a storage layer. There are a couple of patterns used for this. One is Event Sourcing, where we store each state change our service makes in a topic, which can be replayed in full. The second approach optimizes this to only keep the latest event, for each key, using a compacted topic (discussed above). The final approach is to hold both, linked via a Kafka Streams process. This gives you the best of both worlds.
- Одно из самых больших различий между Kafka и другими системами обмена сообщениями заключается в том, что его можно использовать в качестве уровня хранения. Для этого используется пара шаблонов. Одним из них является Event Sourcing, где мы сохраняем каждое изменение состояния, которое делает наша служба, в теме, которую можно воспроизвести полностью. Второй подход оптимизирует это, чтобы сохранить только последнее событие для каждого ключа, используя сжатую тему (обсуждалось выше). Последний подход состоит в том, чтобы удерживать и то, и другое, связанное через процесс Kafka Streams. Это дает вам лучшее из обоих миров.
- Event Sourcing is just the observation that events (i.e., state changes) are a core element of any system. So, if they are stored, immutably, in the order they were created in, the resulting event log provides a comprehensive audit of exactly what the system did. What's more, we can always rederive the current state of the system by rewinding the log and replaying the events in order.

event sourcing + compacted topic итоговый подход

- event sourcing == означает что храним сообщения вечно как единые источник правды source of truth
- compacted topic = загружаемся для скорости из этих сжатых топиков (которые получены из единого source of truth)

в кафке можно хранить сотни терабайт

В Kafka можно хранить очень большие наборы данных. В конце концов, он разработан для рабочих нагрузок с «большими данными», а производственные варианты использования объемом более ста терабайт не редкость как в формах на основе хранения, так и в сжатых формах.

kafka = event sourcing = event store = master системой всех событий

...

единий сервер кафки для всей организации

- What does it mean that our log is unified
Что значит, что наш журнал единый? Это означает, что у нас есть единое развертывание этой технологии в нашей компании (или подразделении, или где-то еще), с несколькими приложениями, отправляющими события и считающими события из него.
- Kafka спроектирован так, чтобы один кластер мог служить центральной магистралью данных для большой организации.

трактовка 1. см что такое event sourcing и command sourcing

onenote:///C:\Users\trans\Qsync\vova_from_onenote\tf_algonote_v1\SYSTEMDESIGN\KAFKA_\ REST.one#command%20sourcing§ion-id={A351751F-B444-4057-93FB-D671AB1B0B21}&page-id={9EAFE1E3-2C7E-4341-A6F3-686031136509}&end

- Данные о событиях записывают то, что происходит, а не то, чем вещи являются на самом деле (те не просто последовательность данных). В веб-системах это означает ведение журнала активности пользователей, а также события на уровне машины и статистику, необходимые для надежной работы и мониторинга оборудования центра обработки данных. Люди склонны называть это «данными журнала», поскольку они часто записываются в журналы приложений, но это путает форму с функцией. Эти данные лежат в основе современного Интернета: в конце концов, состояние Google создается за счет конвейера релевантности, основанного на кликах и показах, то есть событиях.

event sourcing - popular pattern for event logging

- те это когда в логе сохраняем каждый факт
- A popular pattern for event logging is event sourcing, in which we capture the state change — triggered by a command or request — as a new event to be stored in the event log. These events represent the fact that something has happened (i.e., OrderCreated, PaymentAuthorized, or PaymentDeclined).
- One benefit of using event sourcing is that it allows the aggregate to cache the dataset—the latest snapshot—in memory, instead of having to reconstitute it from durable storage every request
- This pattern is often referred to as a Memory Image, and it helps to avoid the infamous Object-Relational Impedance Mismatch.
- Each event-sourced aggregate usually has an event stream through which it is publishing its events to the rest of the world—for example, such as through Apache Kafka. The event stream can be subscribed to by many different parties, for different purposes

Perform state changes only by applying events. Make them durable by storing the events in a log (this pattern was described in 2005 by Martin Fowler)

- The Event-Sourcing pattern turns the destructive update of persistent state into a nondestructive accumulation of information by recognizing that the full history of an object's state is represented by the change events it emits. Шаблон Event-Sourcing превращает деструктивное обновление постоянного состояния в неразрушающее накопление информации, признавая, что полная история состояния объекта представлена событиями изменения, которые он испускает.

declaring the event log to be the sole source of truth

- You want your domain objects to retain their state across system failures as well as cluster shard-rebalancing events, and toward this end you must make them persistent

- The source of truth needs to be persistent, and because events are generated in strictly sequential order, you merely need an append-only log data structure to fulfill this requirement.
- This pattern is applicable where the durability of an object's state history is practical and potentially interesting

Изменять состояние можно только путем применения событий

- <https://martinfowler.com/eaaDev/EventSourcing.html>
- Instead of transforming the state-changing events into updates of a single storage location, you can turn things around and make the events themselves the source of truth for your persistent domain objects—hence, the name event-sourcing. Вместо того, чтобы преобразовывать события, изменяющие состояние, в обновления одного места хранения, вы можете изменить ситуацию и сделать сами события источником истины для ваших постоянных объектов домена - отсюда и название « event-sourcing »

события никогда не удаляются

- Источники событий обычно неприменимы в случаях, когда желательно удалить события из журнала. Мало того, что вся концепция построена на представлении неизменяемых фактов, но это желание обычно возникает, когда постоянное состояние не имеет значения в бизнес-области - например, при использовании PersistentActor в качестве устойчивой очереди сообщений

что такое event sourcing по Фаулеру: Event Sourcing ensures that all changes to application state are stored as a sequence of events.

- Вместо того, чтобы сохранять текущее состояние в приложении непосредственно поле за полем в таблице в базе данных, перезагружая его при необходимости и перезаписывая его последующими изменениями, сохраняется хронологически упорядоченный список событий, который затем можно использовать для восстановления текущего при необходимости записать в память.
- Сохраненные события описывают не только текущее состояние, но и то, как это состояние было достигнуто.
- В любой момент можно восстановить любое состояние из прошлого, воспроизведя события только до определенного момента времени.
- Возможно использование источника событий для обработки некорректной обработки предыдущих событий или прихода отложенного события.

трактовка 2: DDD domain events

события всегда публикуются при создании и обновлении агрегата

- С одной стороны, такая логика довольно проста: каждый метод агрегата, который инициализирует или меняет его состояние, возвращает список событий.
- этот шаблон сохраняет все события каждого агрегата внутри так называемого хранилища событий

трактовка 3

- Pattern: Event sourcing: Persist an aggregate as a sequence of domain events that represent state changes
- Event sourcing is a different way of structuring the business logic and persisting aggregates.
- event sourcing - это способ сохранения состояния агрегата
- Порождение событий — это еще один способ структурирования бизнес-логики и сохранения агрегатов.
- Агрегаты сохраняются в виде последовательности событий, каждое из которых представляет изменение состояния агрегата.
- Приложение воссоздает текущее состояние, воспроизводя записанные события
- шаблон «Порождение событий» сохраняет агрегат в виде событий, из которых затем восстанавливается его текущее

состояние

также надежный механизм для публикации и доставки событий

- Сохранение события в хранилище по своей сути атомарная операция. Нам нужно реализовать механизм для доставки всех сохраненных событий заинтересованным потребителям

Это позволяет микросервису всегда восстанавливать свое состояние по событиям.

- Следовательно, состояние может быть отброшено и воссоздано из событий, если они доступны без каких-либо пропусков.

трактовка 4

Event Sourcing управляет состоянием микросервиса через поток событий, из которых можно определить текущее состояние.

- Event Sourcing administrates the state of a microservice via a stream of events from which the current state can be deduced.

(а) event - это изменение состояния агрегата

- События представляют изменения состояния
- доменные события — это механизм уведомления подписчиков об изменениях, вносимых в агрегаты.
- События могут содержать минимальную информацию, такую как ID агрегата, или быть расширены с помощью данных, которые могут пригодиться типичному потребителю.
 - Например, при создании заказа сервис Order может опубликовать событие OrderCreated. Содержимое OrderCreated может ограничиваться полем orderId.
 - Или содержать весь заказ, чтобы его потребителю не нужно было извлекать эти данные из сервиса Order.
 - Факт публикации события и его содержимое определяются тем, что именно нужно потребителю. Но в случае с порождением событий эта ответственность ложится на сам агрегат.

(б) в сага-хореографии иVENTЫ должны генерироваться агрегатом, даже если состояние не меняется

- Проблема использования событий для хореографии повествований состоит в их двойном назначении. В шаблоне «Порождение событий» они описывают изменение состояния, но в хореографии повествований должны генерироваться агрегатом, даже если состояние не меняется. Например, если обновление агрегата нарушает бизнесправило, тот должен сгенерировать событие, чтобы сообщить об ошибке. Еще более серьезная проблема может возникнуть, когда участнику повествования не удается создать агрегат, в этом случае вернуть ошибку попросту некому

события генерируются всегда, Каждый раз, когда агрегат меняет свое состояние
это жесткое правило, даже если это событие никому ненужно и нет ни одного потребителя события (то все равно
домен должен его сформировать и опубликовать)

- При использовании рассмотренного подхода события генерируются всегда.
- Каждое изменение состояния агрегата, включая его создание, представлено доменным событием. Каждый раз, когда агрегат меняет свое состояние, он обязан сгенерировать событие.
- Например, агрегат Order должен генерировать OrderCreated во время своего создания, а также события вида Order* при каждом своем обновлении
- Это требование куда более жесткое, чем то, что мы видели ранее, когда агрегат генерировал только те события, которые были интересны потребителям

Чтобы интегрировать порождение событий в существующее приложение, вы должны переписать бизнес-логику

это событийный подход к реализации бизнес-логики и сохранению агрегатов

- Агрегат хранится в базе данных в виде цепочки событий.
- Каждое событие представляет изменение его состояния.
- Бизнес-логика агрегата структурирована вокруг требования о генерации и потреблении этих событий
- переписать бизнес логику на event sourcing - как высокий порог вхождения и основной недостаток event sourcing

Методы агрегата полностью полагаются на события

- Чтобы обработать запрос и обновить агрегат, бизнес-логика вызывает командный метод из его корня. В традиционных приложениях командные методы обычно проверяют свои аргументы и затем обновляют одно или несколько полей агрегата. В приложениях, основанных на порождении событий, командные методы должны генерировать события. Результатом вызова командного метода агрегата является последовательность событий, описывающая изменения, которые нужно внести в состояние (рис. 6.4). Эти события хранятся в базе данных и применяются к агрегату для его обновления
- Обязательные генерация и применение событий требуют хоть и прямолинейной, но реструктуризации бизнес-логики. Порождение событий превращает командный метод в два и более метода. Первый из них принимает командный объект, который представляет запрос, и определяет, какое изменение состояния нужно выполнить. Он проверяет свои аргументы и, не меняя состояния агрегата, возвращает список событий, описывающих переход состояния. Если команду не удается выполнить, этот метод обычно генерирует исключение.
- Остальные методы принимают в качестве параметра событие определенного типа и обновляют агрегат. Для каждого события предусмотрен свой метод. Важно отметить, что ни один из этих методов не может отказать, поскольку событие представляет собой изменение состояния, которое уже произошло. В каждом случае агрегат обновляется на основе события.

Эволюция структуры событий

- Шаблон «Порождение событий», по крайней мере на концептуальном уровне, сохраняет события навсегда. Это палка о двух концах. С одной стороны, приложение получает журнал аудита с записями об изменениях, точность которых гарантируется. Это также позволяет приложению воссоздать любое предыдущее состояние агрегата. Но с другой — может возникнуть проблема, поскольку структура многих событий со временем меняется
- Существует вероятность того, что приложению придется иметь дело с несколькими версиями событий. Например, у сервиса, загружающего агрегат Order, может возникнуть необходимость в сохранении разных версий событий. Точно так же потребитель потенциально может видеть несколько версий
- При использовании порождения событий структура событий (и снимков!) будет меняться со временем. Поскольку события хранятся вечно, агрегатам, возможно, придется сворачивать их с учетом нескольких версий структуры. Существует реальный риск того, что агрегаты станут слишком раздутыми из-за кода, предназначенного для разных версий

сохраняет историю агрегатов

- преимущества этой методики, включая **сохранение полной истории изменений агрегата**
- Порождение событий имеет несколько важных преимуществ. Например, оно сохраняет историю агрегатов, которая может пригодиться для аудита или соблюдения нормативно-правовых норм. Оно также выполняет надежную публикацию доменных событий, что особенно полезно в микросервисной архитектуре.

event sourcing основан на концепции доменных событий (те событий домена, те событий агрегата)

- Порождение событий основано на концепции доменных событий и работает совсем иначе.
- Оно сохраняет каждый агрегат в базе данных, так называемом хранилище событий, в виде последовательности событий.
- Возьмем в качестве примера агрегат Order. Вместо сохранения каждого экземпляра Order в виде строки таблицы ORDER порождение событий помещает каждый агрегат Order в таблицу EVENTS, используя одну или несколько строк (рис. 6.2). Каждая строка — это доменное событие, такое как Order Created, Order Approved, Order Shipped и т. д.

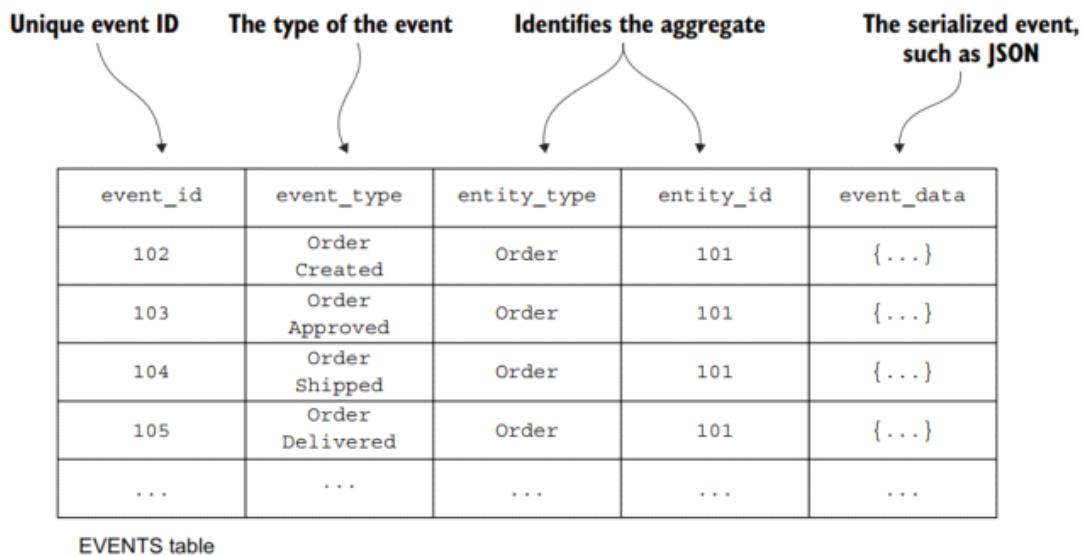


Figure 6.2 Event sourcing persists each aggregate as a sequence of events. A RDBMS-based application can, for example, store the events in an EVENTS table.

как загрузить агрегат из event store

- Он создает экземпляр класса и проходит по событиям, вызывая метод агрегата applyEvent(). Если вы знакомы с функциональным программированием, то, наверное, догадались, что это операция свертывания

store by retrieving its events and replaying them. Specifically, **loading an aggregate** consists of the following three steps:

- 1 Load the events for the aggregate.
- 2 Create an aggregate instance by using its default constructor.
- 3 Iterate through the events, calling `apply()`.

For example, the Eventuate Client framework, covered later in section 6.2.2, uses code similar to the following to reconstruct an aggregate:

```
Class aggregateClass = ...;
Aggregate aggregate = aggregateClass.newInstance();
for (Event event : events) {
    aggregate = aggregate.applyEvent(event);
}
// use aggregate...
```

An aggregate is created using the following steps:

- 1 Instantiate aggregate root using its default constructor.
- 2 Invoke `process()` to generate the new events.
- 3 Update the aggregate by iterating through the new events, calling its `apply()`.
- 4 Save the new events in the event store.

An aggregate is updated using the following steps:

- 1 Load aggregate's events from the event store.
- 2 Instantiate the aggregate root using its default constructor.
- 3 Iterate through the loaded events, calling `apply()` on the aggregate root.
- 4 Invoke its `process()` method to generate new events.
- 5 Update the aggregate by iterating through the new events, calling `apply()`.
- 6 Save the new events in the event store.

известные недостатки event sourcing

Drawbacks of event sourcing

Event sourcing isn't a silver bullet. It has the following drawbacks:

- It has a different programming model that has a learning curve.
- It has the complexity of a messaging-based application.
- Evolving events can be tricky.
- Deleting data is tricky.
- Querying the event store is challenging.

event store - Приложение, использующее порождение событий, хранит события в отдельном хранилище

- *Event Store*—A .NET-based open source event store developed by Greg Young, an event sourcing pioneer (<https://eventstore.org>).
- *Lagom*—A microservices framework developed by Lightbend, the company formerly known as Typesafe (www.lightbend.com/lagom-framework).
- *Axon*—An open source Java framework for developing event-driven applications that use event sourcing and CQRS (www.axonframework.org).
- *Eventuate*—Developed by my startup, Eventuate (<http://eventuate.io>). There are two versions of Eventuate: Eventuate SaaS, a cloud service, and Eventuate Local, an Apache Kafka/RDBMS-based open source project.

Хранилище событий — это гибрид базы данных и брокера сообщений

- Оно ведет себя как БД, потому что у него есть API для вставки и извлечения событий агрегата по первичному ключу.
- Но оно похоже и на брокер сообщений, потому что у него есть API, который позволяет подписываться на события

нужно гарантировать порядок событий

- Микросервис получает событие для начального баланса, но еще не получил регистрацию пользователя. Микросервис не может кредитовать первоначальный баланс, потому что клиент не был создан в системе. Если регистрация клиента происходит позже, первоначальный баланс необходимо будет выполнить еще раз
- Эта проблема может быть решена, если можно гарантировать порядок событий.

продьюсер может отправлять события в разное время

Самый простой вариант - отправить событие после того, как произошло фактическое действие.

- Таким образом, производитель сначала обрабатывает заказ, прежде чем сообщить другим микросервисам о заказе с помощью события. В этом случае, однако, производитель может изменить данные в базе данных и не отправлять событие, потому что, например, он терпит неудачу до отправки события.

производитель также может отправлять события до фактического изменения данных

- Поэтому, когда поступает новый заказ, производитель отправляет событие перед изменением данных в локальной базе данных. В этом нет особого смысла, потому что события на самом деле являются информацией о событии, которое уже произошло. Наконец, во время действия может возникнуть ошибка. Если это произойдет, значит, событие уже было отправлено, хотя действие не было выполнено.
- Отправка событий до фактического действия также имеет тот недостаток, что фактическое действие задерживается. Сначала отправляется событие, что занимает некоторое время, и только после отправки события может быть выполнено действие. Таким образом, действие откладывается на время, необходимое для отправки события. Это может привести к снижению производительности.

event sourcing, a technique that was developed in the domain-driven design (DDD) community

источники событий подразумевают сохранение всех изменений состояния приложения в виде журнала изменений

- Источники облегчают развитие приложений, помогают при отладке, упрощая понимание того, что и почему происходит, и защищают от ошибок приложений

важно записывать действия пользователя как неизменные события

- , а не результат этих действий в изменяемой базе.
- Например, при сохранении события «студент отказался от зачисления на курс» ясно показана цель действия, и это выражено в нейтральной форме, тогда как по его результату «запись была удалена из таблицы зачислений, причина отмены добавлена в таблицу обратной связи со студентом» можно сделать различные предположения о том, как будут использоваться данные позднее. Если вводится новая функция приложения — например, «место предложено следующему претенденту из списка ожидания», — то источники событий позволяют легко связать новый результат с существующим событием.

Источники событий похожи на хронологическую модель хранения данных

- а также на журнал событий

Специализированные базы данных, такие как Event Store [46], были разработаны для поддержки приложений с помощью источников событий,

- но технология в целом не зависит от какого-либо конкретного инструмента. Для создания приложений в этом стиле можно использовать также обычную БД или брокер сообщений на основе журналирования.

обычно пользователи ожидают увидеть текущее состояние системы, а не историю его изменений

- Например, в интернет-магазине покупатель ожидает увидеть текущее содержимое своей корзины, а не список всех изменений, которые он когда-либо в ней делал
- Таким образом, приложения, применяющие источники событий, должны брать журнал событий (представляющий данные, записанные в систему) и преобразовывать их в состояние, подходящее для представления пользователю (способ чтения данных из системы [47]). Это преобразование может задействовать произвольную логику, но оно должно быть однозначным, чтобы при многократном выполнении генерировать по данным из журнала событий одинаковое состояние приложения.

воспроизведение журнала событий позволяет восстановить текущее состояние системы

Приложения, использующие источники событий, как правило, имеют некий механизм для хранения моментальных снимков текущего состояния системы, которое получают из журнала событий, поэтому им не нужно многократно обрабатывать весь журнал.

- Однако данная оптимизация касается только производительности — она позволяет ускорить чтение и восстановление после сбоев; система должна постоянно хранить все необработанные события и при необходимости обрабатывать журнал целиком

Всякий раз, когда состояние изменяется, это является результатом изменившихся событий.

- Например, список свободных мест есть результат операций бронирования, ранее обработанных системой, текущий баланс на счету — результат кредитных и дебетных операций со счетом, а график времени отклика для веб-сервера — результат сбора показаний времени отклика для всех веб-запросов, которые произошли за определенное время
- Независимо от того, как именно изменяется состояние, всегда существует последовательность событий, вызвавших эти изменения. Даже если операция была сделана, а потом отменена, факт остается фактом: события произошли. Основная идея заключается в том, что изменяемое состояние и список событий, открытый только для дополнения, не противоречат друг другу: это две стороны одной и той же медали. Журнал изменений отражает эволюцию состояния системы с течением времени

Связь между текущим состоянием приложения и потоком событий

$$state(now) = \int_{t=0}^{now} stream(t) dt \quad stream(t) = \frac{d state(t)}{dt}$$

Figure 11-6. The relationship between the current application state and an event stream.

Долговременное хранение журнала изменений приводит к простому результату: состояние является воспроизводимым.

- Если журнал событий является системой записи и из него можно получить любое изменяемое состояние, то будет легче рассуждать о потоке данных через систему

Неизменяемые события хранят больше информации, чем только текущее состояние

- Например, в интернет-магазине клиент может добавить товар в корзину а затем удалить его оттуда. Хотя с точки зрения выполнения заказа второе событие отменяет первое, в целях аналитики может быть полезно знать, что клиент собирался купить данный товар, но затем отказался. Вероятно, он захочет купить его позже, или стоит предложить ему замену. Эта информация записывается в журнал событий, но будет потеряна в БД, так как в ней из корзины будут удалены соответствующие элементы при удалении товара

event store

27 января 2021 г. 21:21

версия по Eberhard Wolf

event store - как отдельный микросервис

- Хранилище событий может быть частью микросервиса, который принимает события и сохраняет их в своем собственном хранилище событий. В качестве альтернативы инфраструктура не только отправляет события, но и сохраняет их.
- должна хранить старые события, она должна обрабатывать значительные объемы данных. Следовательно, если старые события отсутствуют, состояние микросервиса больше не может быть восстановлено по событиям.
- хранить данные в центральном микросервисе, который доступен для всех. В результате все микросервисы будут получать самые последние данные. Однако в случае сбоя центрального микросервиса все остальные микросервисы будут недоступны

event store - на инфраструктуре в очереди

- На первый взгляд, лучше, если инфраструктура хранит события, поскольку упрощает внедрение микросервисов. В таком случае хранилище событий будет реализовано в очереди событий.
- При хранении событий в инфраструктуре необходимо найти модель события, которая удовлетворяет всем микросервисам. В конце концов, каждый микросервис представляет собой отдельный ограниченный контекст со своей моделью предметной области, поэтому найти общую модель сложно.
- такие подходы, как event sourcing, возможны только в том случае, если каждая микросервис хранит саму историю событий. Кафка, с другой стороны, может сохранять записи постоянно. Kafka также может обрабатывать большие объемы данных и может быть распределена по нескольким серверам

event store распределен по всем микросервисам

- Если каждый микросервис хранит события в своем собственном хранилище событий, микросервис может хранить все соответствующие данные в событии, которые микросервис мог получить из разных источников
- В конечном счете, событие можно рассматривать как своего рода репликацию данных на нескольких микросервисах. Однако, в отличие от полной репликации данных между узлами баз данных NoSQL, каждый микросервис может по-разному реагировать на событие и может использовать только части данных.
- Микросервисы, возможно, еще не получили некоторые события, поэтому данные могут быть противоречивыми. Тем не менее, система может обрабатывать запросы с использованием локальных данных и поэтому доступна даже в случае сбоя других систем

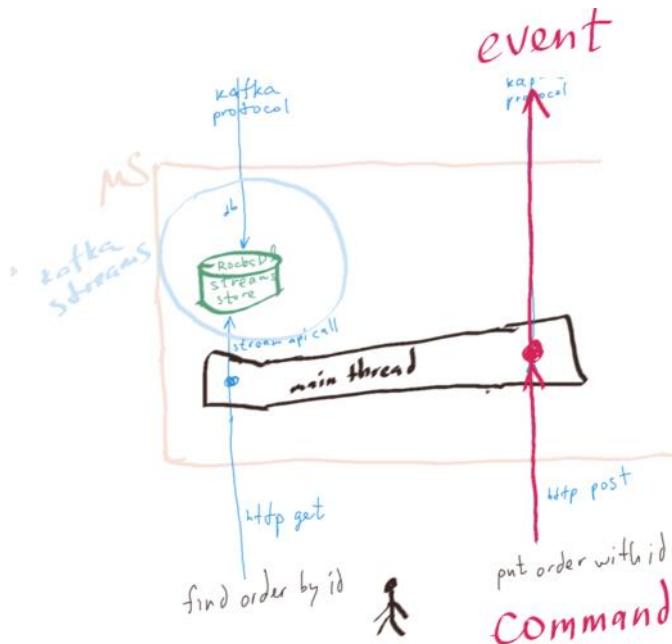
command sourcing

2 января 2021 г. 7:47

первое что надо сделать при попадании команды в систему это преобразовать ее в event и занести в event store (query сохранять не обязательно)

- так как все что происходит с системой должно учитываться

? или же сначала применить команду к агрегату и затем забрать логи БД и выпустить event



Command Sourcing

- Command Sourcing - это, по сути, вариант Event Sourcing, но он применяется к событиям, которые поступают в службу, а не через события, которые она создает.
- Когда совершается покупка (1), Command Sourcing диктует, что запрос заказа должен быть немедленно сохранен как событие в журнале, прежде чем что-либо произойдет (2). Таким образом, если что-то пойдет не так, сервис можно будет перезапустить и воспроизвести, например, для восстановления после повреждения.
- Command Sourcing lets us record our inputs, which means the system can always be rewound and replayed. Event Sourcing records our state changes, which ensures we know exactly what happened during our system's execution, and we can always regenerate our current state (in this case the contents of the database) from this log of state changes

Command Sourcing обеспечивает повторную обработку command от источников, в то время как Event Sourcing повторно извлекает текущее состояние службы только для post-processed ивентов

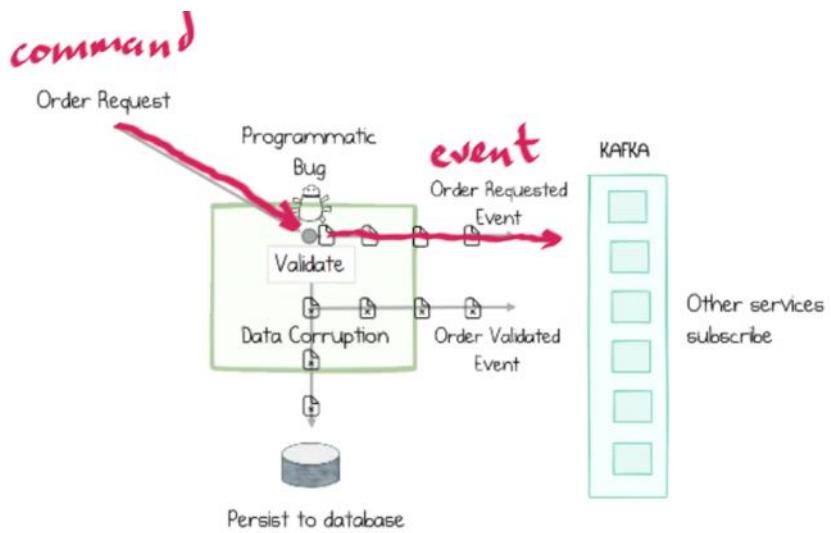


Figure 7-3. Adding Kafka and an Event Sourcing approach to the system described in Figure 7-2 ensures that the original events are preserved before the code, and bug, execute

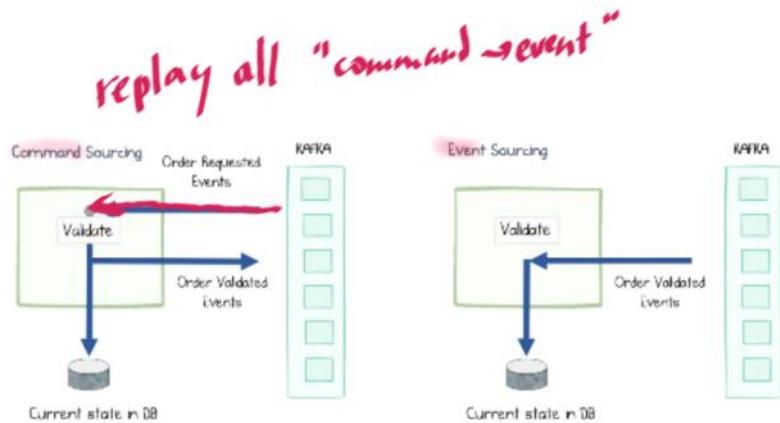


Figure 7-4. Command Sourcing provides straight-through reprocessing from the original commands, while Event Sourcing rederives the service's current state from just the post-processed events in the log

* event storming

5 января 2021 г. 19:49

после мозгового штурма мы получаем commands, events, aggregates и сиквенс диаграмму

- Commands: CreateOrder, SubmitPayment, ReserveProducts, ShipProducts
- Events: OrderCreated, ProductsReserved, PaymentApproved, PaymentDeclined, ProductsShipped
- Aggregates: Orders, Payments, Inventory

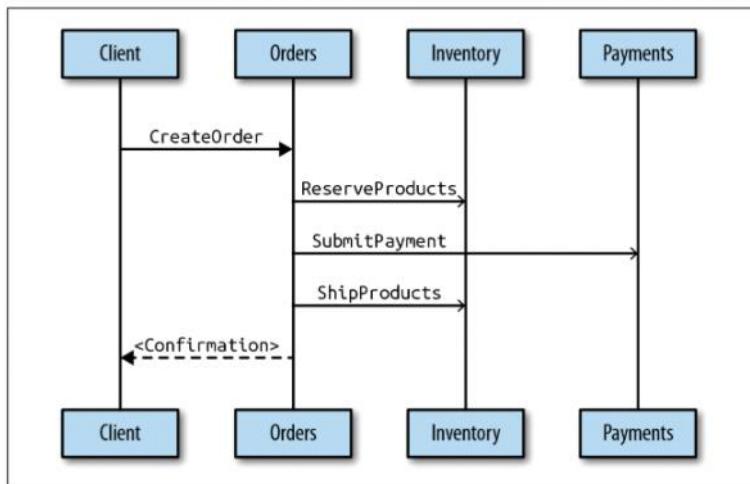


Figure 4-2. The flow of commands in the order management sample use case

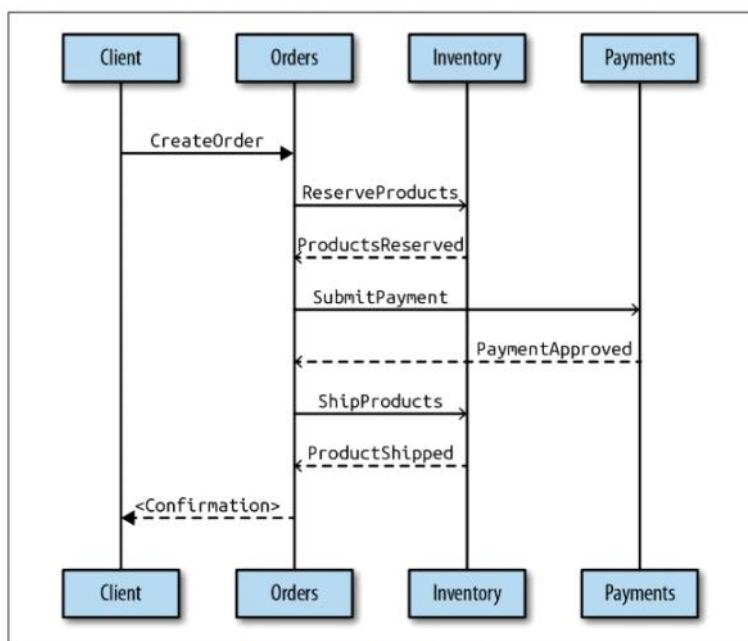


Figure 4-3. The flow of commands and events in the order management sample use case

nouns (the domain objects) to the verbs (the events) in the domain

- те ивенты это глаголы
- Events-First Domain-Driven Design (by Russ Miles)
- Термин « проектирование, ориентированное на конкретные события», было придумано Рассом Майлсом , и это название набора принципов проектирования, которые появились в нашей отрасли за последние несколько лет и оказались очень полезными при построении распределенных систем в большом масштабе. Эти принципы помогают нам сместить фокус с существительных (объектов предметной области) на глаголы (события) в предметной области. Смещение фокуса дает нам лучшую отправную точку для понимания сущности предметной области с точки зрения потока данных и коммуникаций и направляет нас на путь к масштабируемой, управляемой событиями конструкции.

лучше сначала сосредоточится на сквозном процессе а потом уже на структуре

- Объектно-ориентированное программирование (ООП) и более позднее предметно-ориентированное проектирование (DDD) научили нас, что мы должны начинать наши сеансы проектирования, **сосредотачиваясь на вещах - существительных** - в предметной области, как способе поиска объектов предметной области , а затем работать оттуда. Оказывается, у этого подхода есть серьезный недостаток: он заставляет нас слишком рано **сосредотачиваться на структуре** .
- Вместо этого нам следует обратить внимание на то, что происходит - на поток событий - в нашей сфере. Это заставляет нас понять, как изменения распространяются в системе - такие вещи, как шаблоны общения, рабочий процесс, выяснение того, кто с кем разговаривает, кто за какие данные отвечает и т. д. Нам необходимо смоделировать бизнес-домен с точки зрения зависимости данных и коммуникации.
 - Когда вы начинаете моделировать события, это заставляет вас думать о поведении системы, а не о структуре внутри системы.
 - Моделирование событий заставляет вас сосредоточиться во времени на том, что происходит в системе. Время становится решающим фактором системы.

event storming - помогает выявить event fact, data flows и их зависимости

- Это процесс проектирования, в котором вы собираете всех заинтересованных сторон - экспертов в предметной области и программистов - в одну комнату, где они проводят мозговой штурм, используя стикеры, пытаясь найти язык предметной области для событий и команд , исследуя, как они причинно связаны и вызывают реакции.
- Alberto Brandolini's upcoming book Event Storming

think and design in terms of consistency boundaries for the services

- (Pat Helland's paper, "Data on the Outside versus Data on the Inside",)
- (1) **strongly consistent** / unit of consistency / строго согласованный набор данных - относится к одному DDD агрегату
- (2) За пределами границ согласованности агрегата у нас нет другого выбора, кроме как полагаться на **eventual consistency** (конечную согласованность) (те это для событий между агрегатами)

Я счел полезным думать и проектировать с точки зрения **границ согласованности** для сервисов:

1. Не поддавайтесь желанию начать с размышлений о **поведении** службы.
2. Начните с данных - фактов - и подумайте о том, как они связаны и какие зависимости имеют.
3. Определите и смоделируйте ограничения целостности и то, что необходимо гарантировать, с точки зрения предметной области и бизнеса. В этом процессе очень важно проводить интервью с экспертами

- в предметной области и заинтересованными сторонами.
4. Начните с нулевых гарантий для наименьшего возможного набора данных. Затем добавьте самый слабый уровень гарантии, который решит вашу проблему, при этом пытаясь сохранить размер набора данных до минимума.
 5. Пусть *принцип единой ответственности* ([обсуждается в разделе «Единственная ответственность» на стр. 2](#)) будет руководящим принципом.

Цель состоит в том, чтобы попытаться минимизировать набор данных, который должен быть строго согласованным. После того как вы определили существенный набор данных для Сервисов, затем обратитесь к поведению и протоколам для экспонирования данных в процессе взаимодействия с другими службами и системами, определяя наши **unit of consistency (DDD агрегаты)**

- Агрегаты, которые не ссылаются друг на друга напрямую, могут быть повторно разделены и перемещены в кластере для почти бесконечной масштабируемости. как обрисовал Пэт Хелланд в его влиятельной статье «Жизнь за пределами распределенных транзакций» . 5
- За пределами границ согласованности агрегата у нас нет другого выбора, кроме как полагаться на *eventual consistency* (конечную согласованность) . В своей книге « Реализация дизайна, основанного на домене » (Addison-Wesley) Вон Вернон предлагает практическое правило, как думать об ответственности в отношении согласованности данных. Вы должны задать себе вопрос: «А чья работа обеспечивать согласованность данных?» Если ответ таков, что бизнес-логику выполняет сервис, подтвердите, что это можно сделать за один раз.

Unit of failure = unit of consistency

everything that must be consistent is not distributed

- A consistent unit must not fail partially; if one part of it fails, then the entire unit must fail

? Persisting isolated scopes of consistency

- так как нам ненадо делать апдейты и удаления, то это снижает требования к механизму хранения, чтобы он работал только как добавляемый журнал событий

один человек == один микросервис

- человек имеется ввиду клиент делающий заказ

Grouping data and behavior according to transaction boundaries

- Чтобы формализовать то, что мы только что обсудили, трюк состоит в том, чтобы нарезать поведение и сопутствующий набор данных таким образом, чтобы каждый фрагмент предлагал желаемые функции изолированно, и не требовались транзакции, охватывающие несколько фрагментов. Этот метод применяется и подробно обсуждается в литературе по предметно-ориентированному проектированию (DDD)

вся почта будет буквально храниться в этом экземпляре. Это означает, что весь доступ к содержимому электронной почты человека будет осуществляться через этот выделенный экземпляр.

- The example problem of storing a person's email folders in a distributed fashion can be solved by applying the strategy outlined in the previous section. If you want to ensure that emails can be moved without leading to inconsistent counts, then each person's complete email dataset must be managed by one entity. The example application decomposition would have a module for this purpose and would instantiate it once for every person using the system. This does not mean all mail would be literally stored within that instance. It only means all access to a person's email content would be via this dedicated instance

Это нормально, потому что человек на много порядков медленнее, чем компьютер, когда дело касается обработки электронной почты, поэтому вы не столкнетесь с проблемами производительности, ограничив масштабируемость в этом направлении

In effect, this acts like a locking mechanism that serializes access, with the obvious

restriction that an individual person's email cannot be scaled out to multiple managers in order to support higher transaction rates. This is fine, because a human is many orders of magnitude slower than a computer when it comes to processing email, so you will not run into performance problems by limiting scalability in this direction. What is more important is that this enables you to distribute the management of all users' mailboxes across any number of machines, because each instance is independent of all others. The consequence is that it is not possible to move emails between different people's accounts while maintaining the overall email count, but that is not a supported feature anyway.

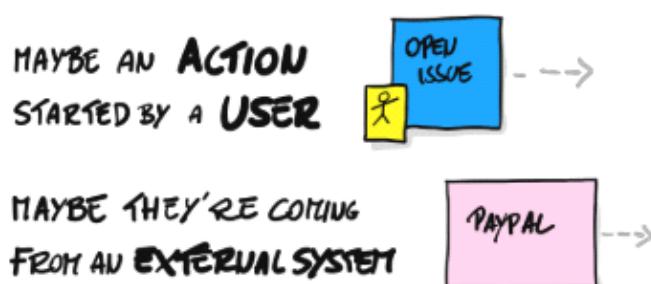
Размещая потоки сообщений в приложении в соответствии с бизнес-процессами, которые вы хотите моделировать

- те взять все могущество бизнес процессов и BPMN
- Размещая потоки сообщений в приложении в соответствии с бизнес-процессами, которые вы хотите моделировать, вы будете явно видеть, кто с кем должен общаться или какой модуль должен будет обмениваться сообщениями с каким другим модулем. Вы также создали иерархическую декомпозицию общей проблемы и, таким образом, получили иерархию надзора, и это покажет вам, какие потоки сообщений с большей или меньшей вероятностью будут прерваны из-за сбоя.

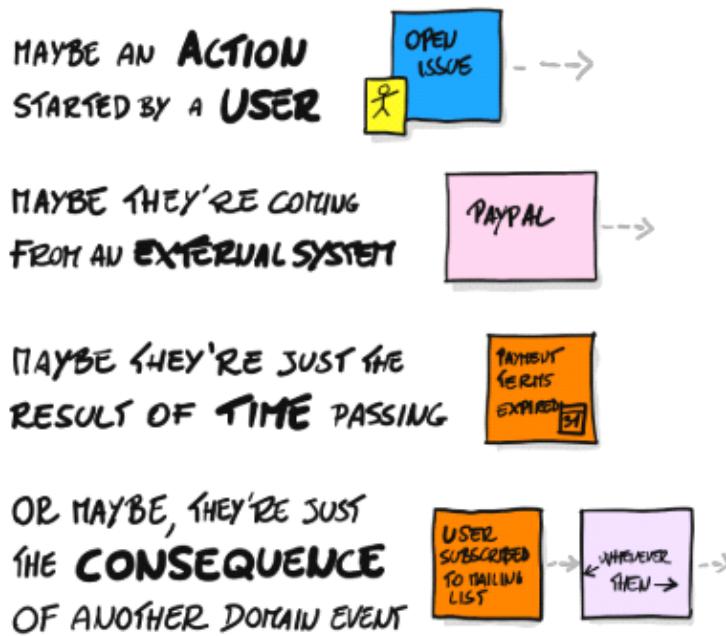
для событий - использование глагола в прошедшем времени

- В источнике нет неявного фильтра: на самом деле они могут происходить по разным причинам:
 - они могут быть следствием какого-либо действия, инициированного пользователем,
 - они могут исходить из какой-то внешней системы ,
 - они могут быть результатом времени
 - они могут быть прямым следствием какого-то другого события.
- Использование глагола в прошедшем времени также заставляет нас исследовать всю область с акцентом на переходы состояний , то есть на точный момент, когда что-то меняется . Это может дать формально иную семантику наших рассуждений. Представьте, что мы собираем информацию о температуре из внешней системы, первым кандидатом на событие DomainEvent может быть повышение температуры . Более пристальный взгляд может показать, что это не совсем то, что произошло; на самом деле нам может потребоваться комбинация регистрации температуры от внешнего источника и приращения температуры, измеренного как следствие, и понять, что первоначальная запись, несмотря на ее правильность, на самом деле была ближе к разговору о погоде, чем к проектированию системы.

WHERE ARE DOMAIN EVENTS COMING FROM?



WHERE ARE DOMAIN EVENTS COMING FROM?



If an event is a record of an occurrence **in the past**, then a command is the expression of intent that a new event will occur **in the future**

- Команда является приказ или указание на конкретное действие , которое будет выполняться в будущем
- почти все программное обеспечение полагается на неявные команды - решение сделать что-либо немедленно сопровождается выполнением этого решения в одном блоке кода.
- Если событие представляет собой запись о произшествии в прошлом , тогда команда является выражением намерения, что новое событие произойдет в будущем
- Грамматически: событие - это глагол в прошедшем времени в изъявительном наклонении , а команда - это глагол в повелительном наклонении .

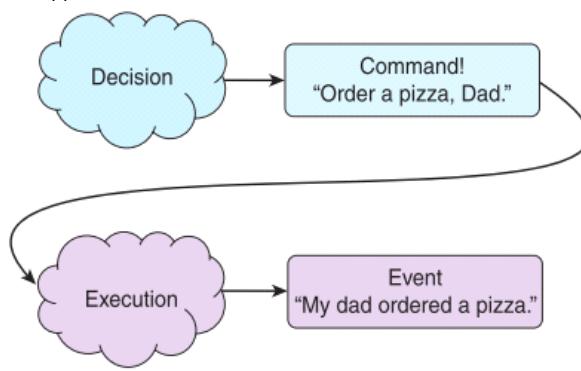
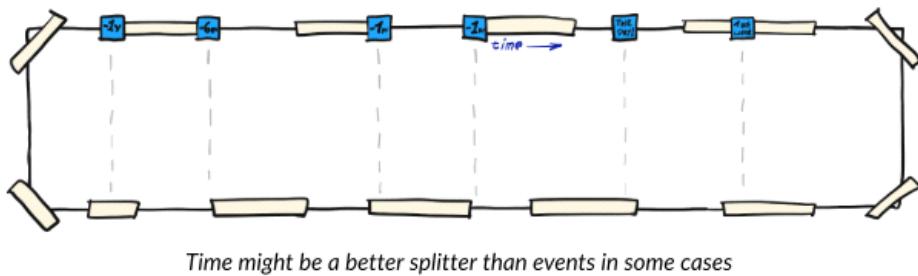


Figure 9.1 A decision produces a command—an order or instruction to do something specific. If that command is then executed, we can record an event as having occurred.

есть такая техника в event storming: где сначала надо наделать стикеров(ивенты, действия) а потом уже упорядочить их на временной шкале

- Временная шкала - отличный способ добиться некоторой степени целостности между пересекающимися повествованиями, но строгое соблюдение временной шкалы не является нашей целью: это просто наш лучший инструмент для обеспечения некоторой последовательности в разговоре.



standard definition of an event as subject-verb-object (t-f: похоже на archimate, похоже на метамодель NLP)

- Как мы должны смоделировать наше первое событие - покупатель просматривает товар во время
- Секрет в том, чтобы понять, что нашему событию уже присуща структура: оно следует грамматическим правилам предложения на английском языке. Для тех из нас, кто плохо разбирается в грамматике английского языка, на рис. 2.7 показаны ключевые грамматические компоненты этого события, выраженные в виде предложения.

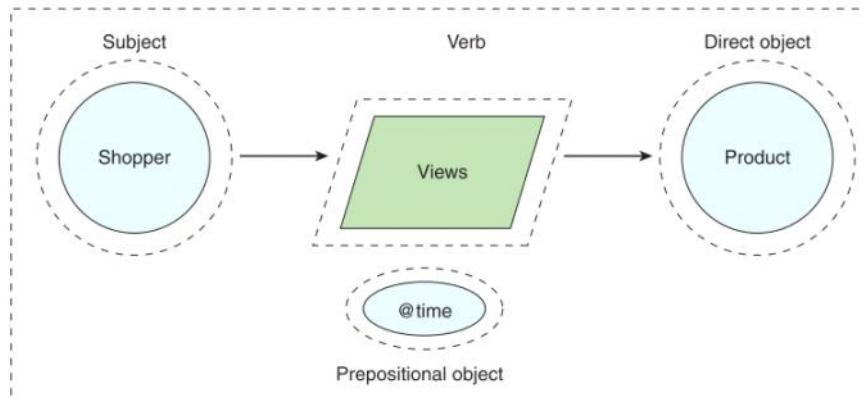


Figure 2.7 Our shopper (subject) views (verb) a product (direct object) at time (prepositional object).

- The “shopper” is the sentence’s *subject*. The subject in a sentence is the entity carrying out the action: *Jane views an iPad at midday*.
- “views” is the sentence’s *verb*, describing the action being done by the *subject*: “*Jane views an iPad at midday*.”
- The “product” being viewed is the *direct object*, or simply *object*. This is the entity to which the action is being done: “*Jane views an iPad at midday*.”
- The time of the event is, strictly speaking, another object—an *indirect object*, or *prepositional object*, where the preposition is “at”: “*Jane views an iPad at midday*.”

представление в виде JSON

Listing 2.1 shopper_viewed_product.json

```
{  
  "event": "SHOPPER_VIEWED_PRODUCT",           ← The event type as a simple  
  "shopper": {                                     string with a verb in the  
    "id": "123",                                simple past tense  
    "name": "Jane",  
    "ipAddress": "70.46.123.145" }                An object representing the  
},                                              shopper who viewed the product  
  "product": {                                     An object representing the  
    "sku": "aapl-001",  
    "name": "iPad" }                            product that was viewed  
},  
  "timestamp": "2018-10-15T12:01:35Z"           ← When the event occurred as an  
}                                              ISO 8601-compatible timestamp
```

Our representation of this event in JSON has four properties:

- event holds a string representing the type of event.
- shopper represents the person (in this case, a woman named Jane) viewing the product. We have a unique id for the shopper, her name and a property called ipAddress, which is the IP address of the computer she is browsing on.
- product contains the sku (stock keeping unit) and name of the product, an iPad, being viewed.
- timestamp represents the exact time when the shopper viewed the product.

примеры

5.2.1 Shopper adds item to cart

The *Shopper adds item to cart* event involves a Nile shopper adding a product to their shopping cart (also known as a *shopping basket*), specifying a quantity of that product as they do this. Let's break out the various components here:

- *Subject*: Shopper
- *Verb*: Adds
- *Direct object*: Item (consisting of product and quantity)
- *Indirect object*: Cart
- *Context*: Timestamp of this event

Figure 5.3 illustrates these components.

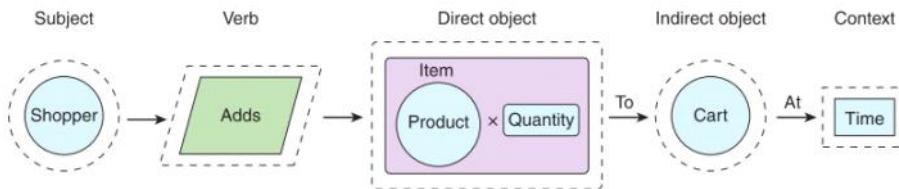


Figure 5.3 Our shopper (again, subject) adds (verb) an item, consisting of a product and its quantity (direct object) to the shopping cart (indirect, aka prepositional, object) at a given time (context).

5.2.2 Shopper places order

This event sounds simple, and it is:

- *Subject*: Shopper
- *Verb*: Places
- *Direct object*: Order
- *Context*: Timestamp of this event

The slight complexity is in modeling the order. This is a complicated entity: it needs to contain an order ID and a total order value, plus a list of items that were purchased in the order; each item should be a product and the quantity of that product ordered.

Putting it all together, we get the event drawn in figure 5.4.

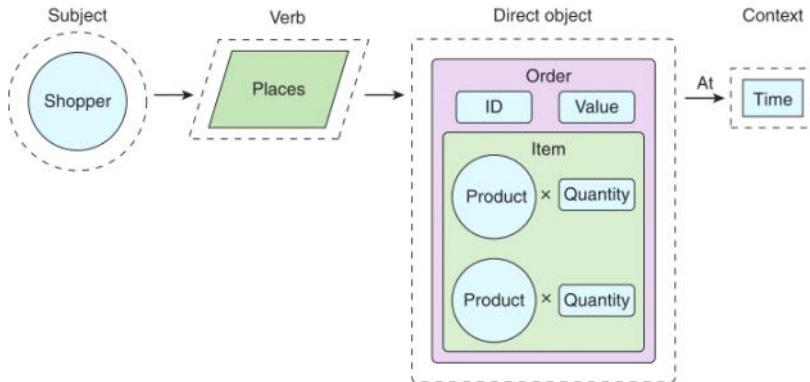


Figure 5.4 Our shopper (always the subject) places (verb) an order (direct object) at a given time (context). The order contains ID and value attributes, plus an array of order items, each consisting of a product and quantity of that product.

5.2.3 Shopper abandons cart

Now we come to our derived event. By *derived*, we mean an event that we are generating ourselves in our stream processing application, as opposed to an incoming raw event that we are simply consuming.

This event looks like this:

- *Subject*: Shopper
- *Verb*: Abandons
- *Direct object*: Cart (consisting of multiple items, each of a product and quantity)
- *Context*: Timestamp of this event

Our fourth and final event for this chapter is illustrated in figure 5.5.

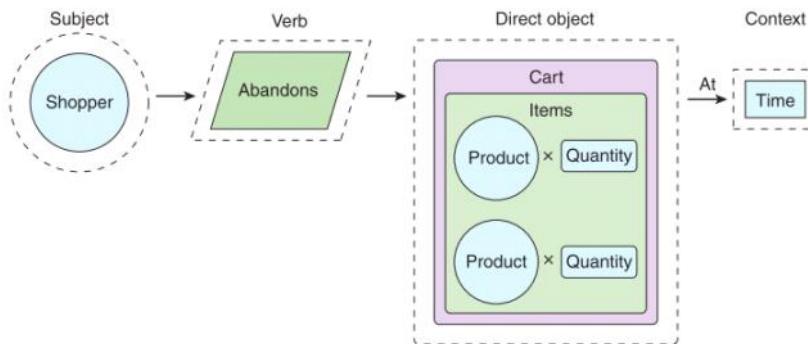


Figure 5.5 Our shopper (subject) abandons (verb) their shopping cart (direct object) at a given time (context). The shopping cart contains an array of order items, each consisting of a product and quantity of that product.

пример 2

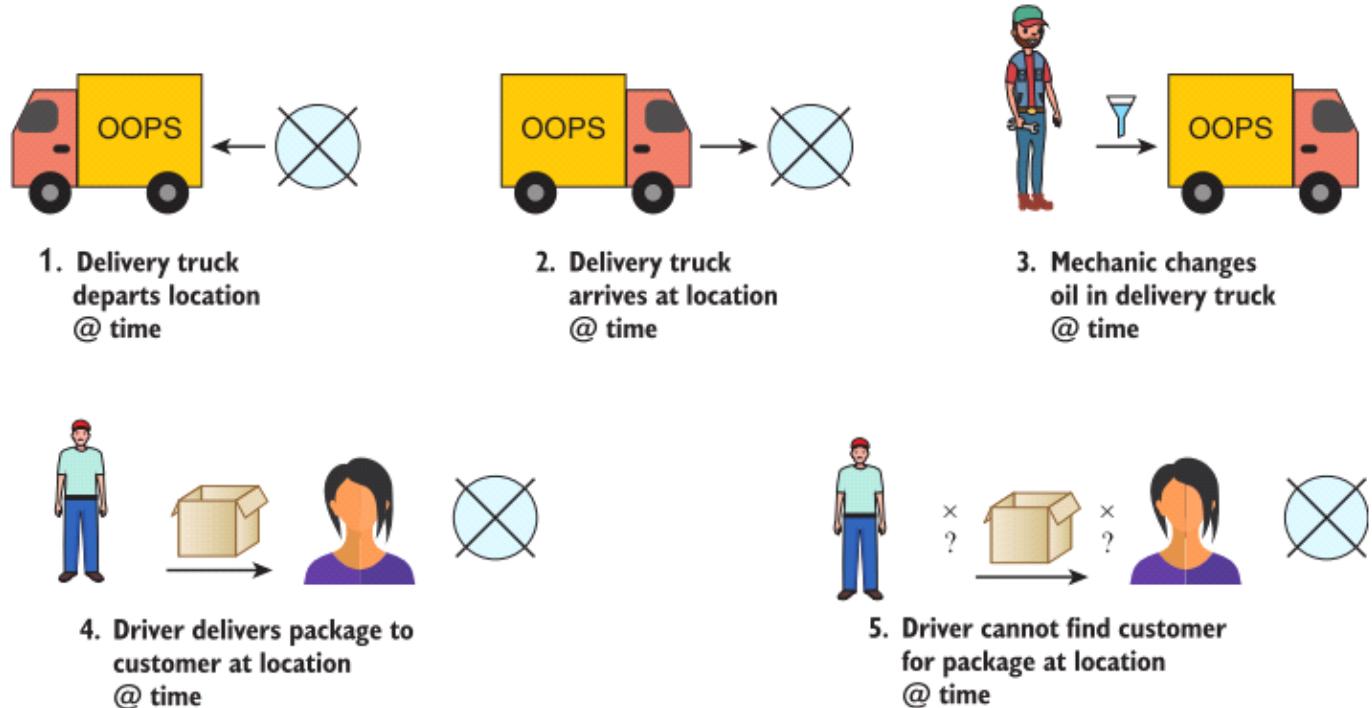


Figure 11.1 The five types of events generated at OOPS are unchanged since their introduction in the previous chapter.

- Delivery truck departs from location at time
- Delivery truck arrives at location at time
- Mechanic changes oil in delivery truck at time

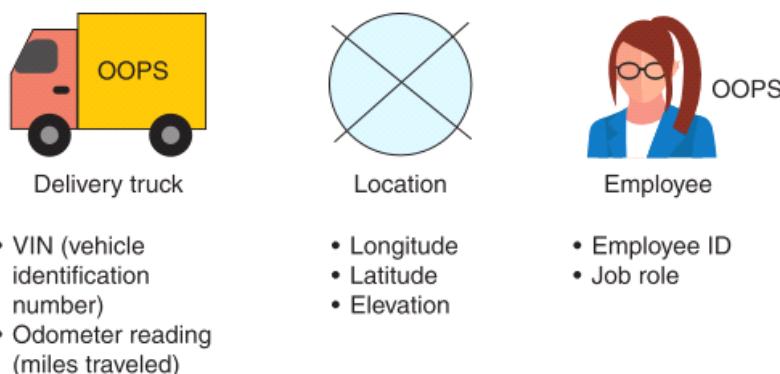


Figure 10.4 The three entities represented in our delivery-truck events are the delivery truck itself, a location, and an employee. All three of these entities have minimal properties—just enough to uniquely identify the entity.

properties:

- A short tag describing the event (for example, TRUCK_DEPARTS or DRIVER_MISSES_CUSTOMER)
- A timestamp at which the event took place
- Specific slots related to each of the entities that are involved in this event

Here is an example of a *Delivery truck departs from location at time* event:

```
{ "event": "TRUCK_DEPARTS", "timestamp": "2018-11-29T14:48:35Z",  
  "vehicle": { "vin": "1HGCM82633A004352", "mileage": 67065 }, "location":  
  { "longitude": 39.9217860, "latitude": -83.3899969, "elevation": 987 } }
```

chapter 6. In the following listing, you can see the JSON schema file for the *Delivery truck departs* event.

Listing 10.1 truck_deperts.json

Similar to the preceding listing, you can see the JSON schema file for the delivery truck departs event.

Listing 10.1 truck_deperts.json

```
{  
  "type": "object",  
  "properties": {  
    "event": {  
      "enum": [ "TRUCK_DEPARTS" ]  
    },  
    "timestamp": {  
      "type": "string",  
      "format": "date-time"  
    },  
    "vehicle": {  
      "type": "object",  
      "properties": {  
        "vin": {  
          "type": "string",  
          "minLength": 17,  
          "maxLength": 17  
        },  
        "mileage": {  
          "type": "integer",  
          "minimum": 0,  
          "maximum": 2147483647  
        }  
      },  
      "required": [ "vin", "mileage" ],  
      "additionalProperties": false  
    },  
    "location": {  
      "type": "object",  
      "properties": {  
        "latitude": {  
          "type": "number"  
        },  
        "longitude": {  
          "type": "number"  
        },  
        "elevation": {  
          "type": "integer",  
          "minimum": -32768,  
          "maximum": 32767  
        }  
      },  
      "required": [ "longitude", "latitude", "elevation" ],  
      "additionalProperties": false  
    }  
  },  
  "required": [ "event", "timestamp", "vehicle", "location" ],  
  "additionalProperties": false  
}
```

- Driver delivers package to customer at location at time
- Driver cannot find customer for package at location at time



Package

• Package ID



Customer

- Customer ID
- Is VIP?

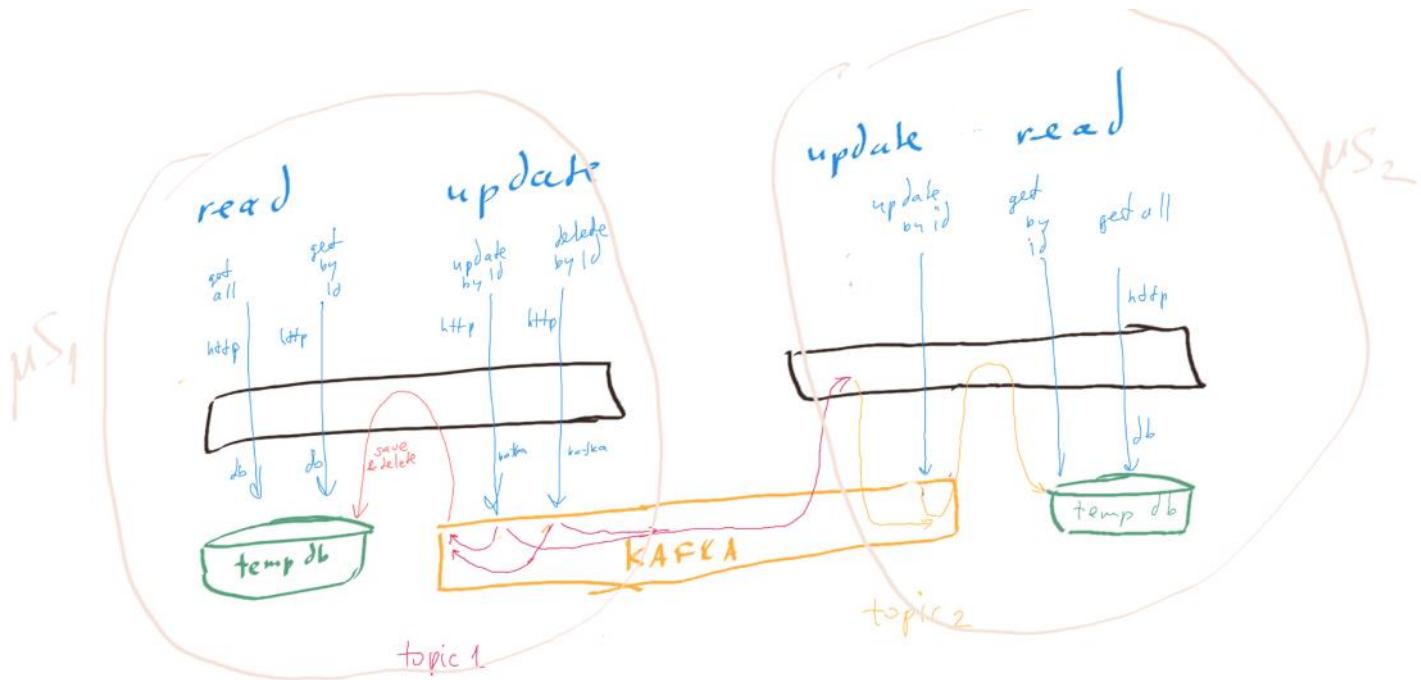
Figure 10.5 The events generated by our delivery drivers involve two additional entities: packages and customers. Again, both entities have minimal properties in the OOPS event model.

event sourcing 1

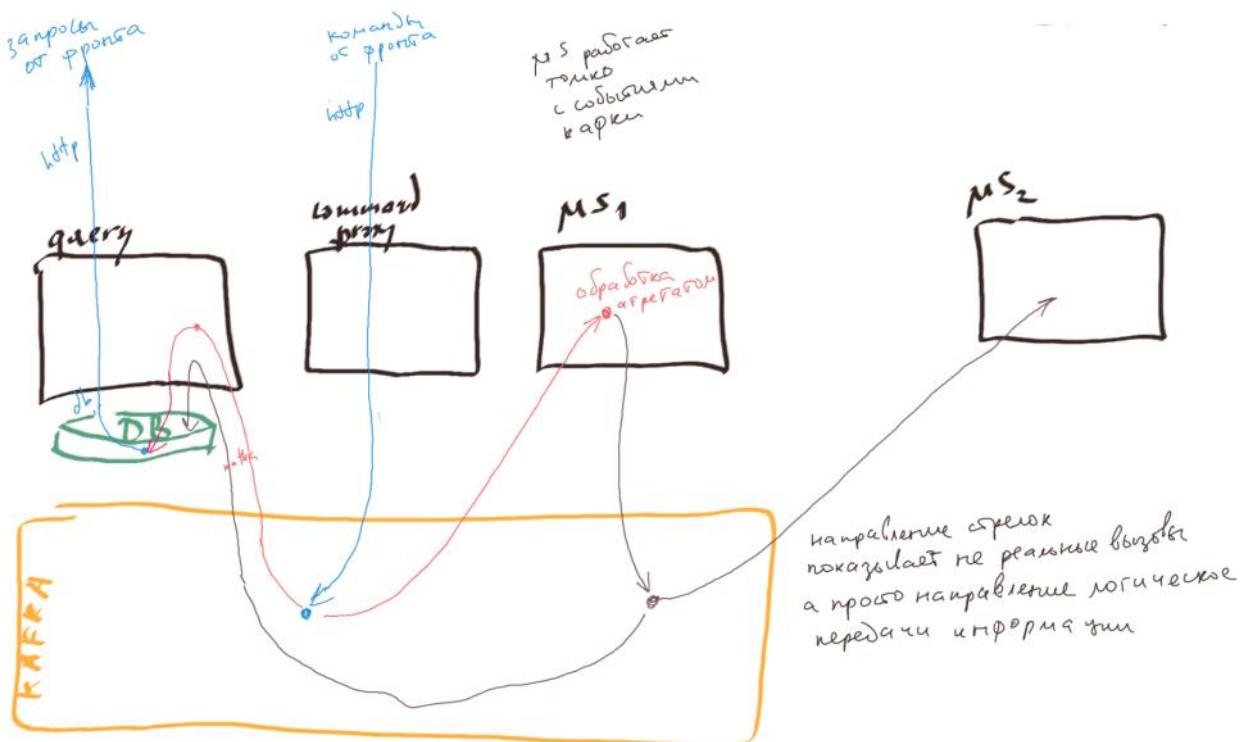
19 декабря 2020 г. 23:30

<https://www.confluent.io/designing-event-driven-systems/>

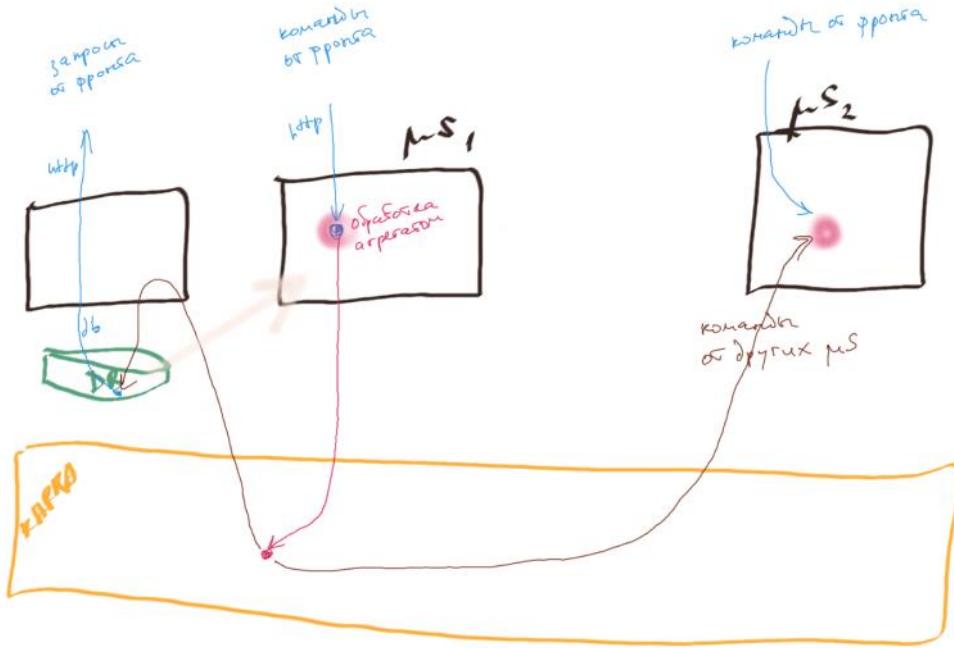
вариант1) микросервис работает только с событиями из кафки, и после обработки также кладет событие обратно в кафку (откуда оно также может забраться этим же микросервисом для своего query store)



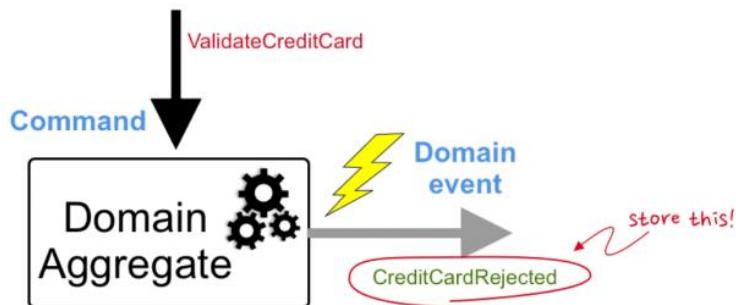
вариант2)



вариант3)

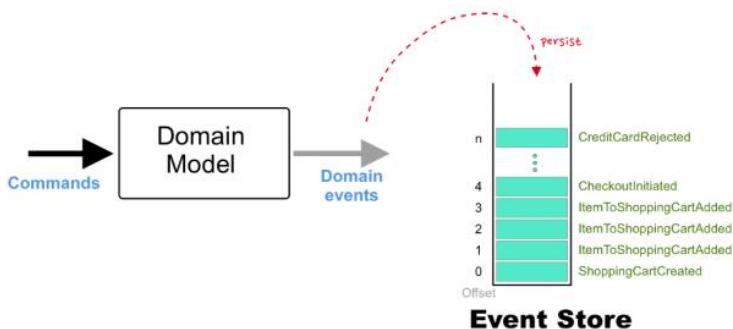


Event Sourcing



- Instead of only working with the state of the world, as it is "now", we want to see how the world changes.
- We introduce the concept of an **event**
- Events** tell us what changed
- A **command** is applied to a **domain aggregate** (or **domain object**)
- A **command**, if successfully applied to the **domain aggregate** always triggers a corresponding **event**
- Let's look at a few samples:
 - AddItemToShoppingCart → ItemToShoppingCartAdded
 - CheckoutShoppingCart → CheckoutInitiated
 - ValidateCreditCard → CreditCardValidated or CreditCardRejected
- Instead of updating the current state (of the domain aggregate) in a database we store the events in a (commit) log

The Event Store

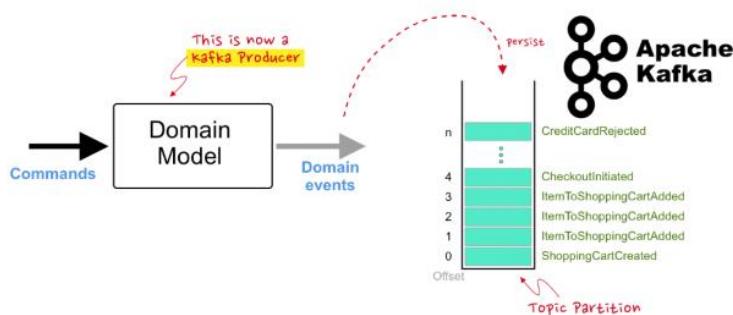


- Events are stored in a structure often called a **commit log**
- A new event is appended **at the end** of the log
- Existing events are immutable, that is, they are never updated or deleted
- The effect of wrong events must be corrected by so called **compensating events**
- The current state** of the world can be achieved by replaying all events
- We can get the **state at any time** in the past by replaying the events up to that particular point in time



Broken down to the Kafka world: All these events got to the same topic and the key is e.g. `shoppingCartId` such as that all events belonging to the same cart go to the same partition and thus retain ordering.

Kafka as Event Store



Of course, when observing the previous slide with the **event store** we should immediately feel familiar. Doesn't this resemble to what Kafka offers us in the first place? And indeed, Apache Kafka is the ideal event store with all the guarantees we expect:

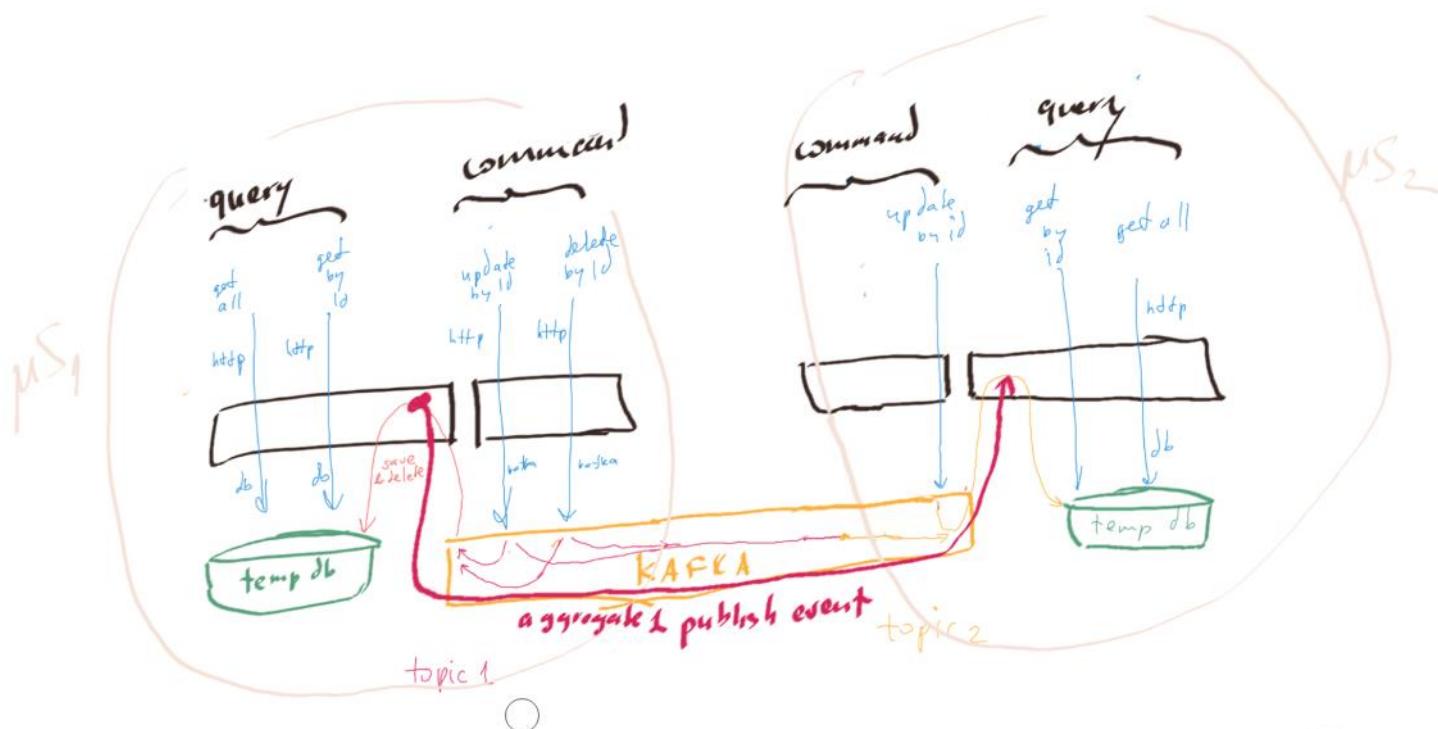
- High volume
- Durability
- Availability
- Append only commit log

Perfect! Here we thus see the connection between the generic event driven architecture and the Confluent Platform driven by Apache Kafka.

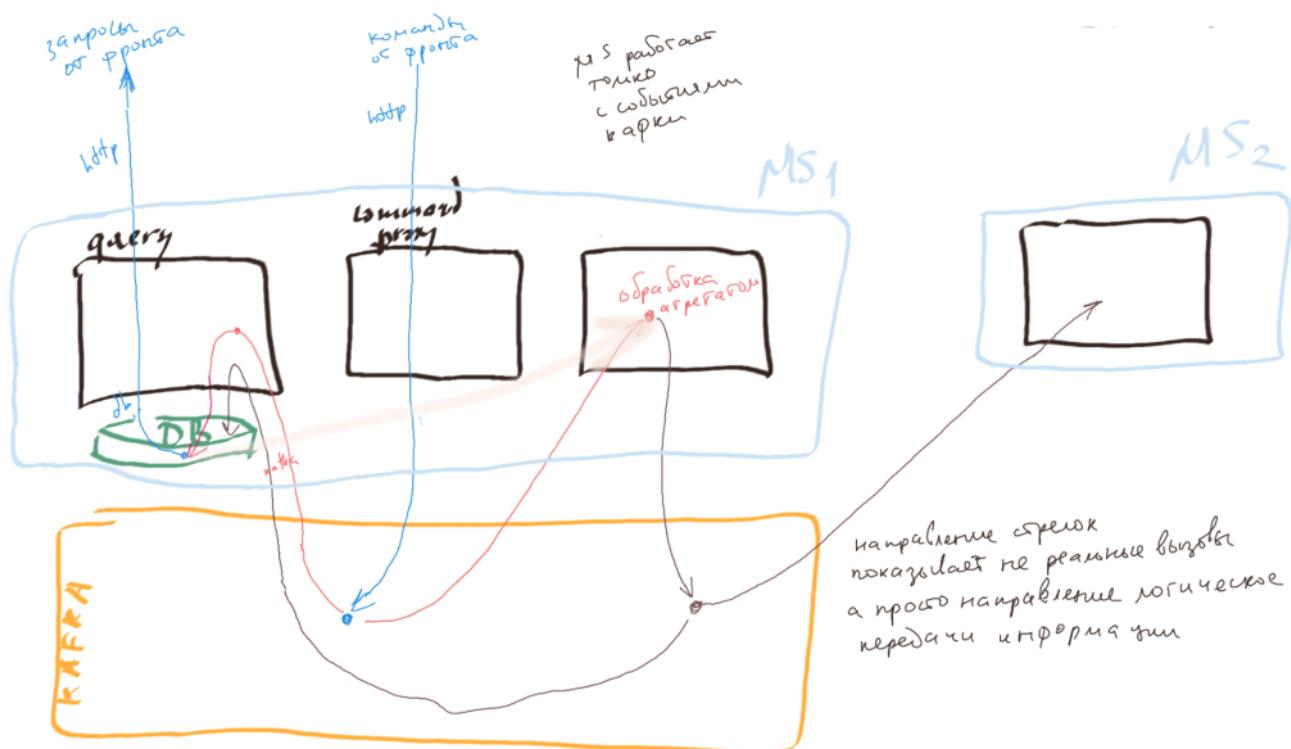
event sourcing 2

19 декабря 2020 г. 23:30

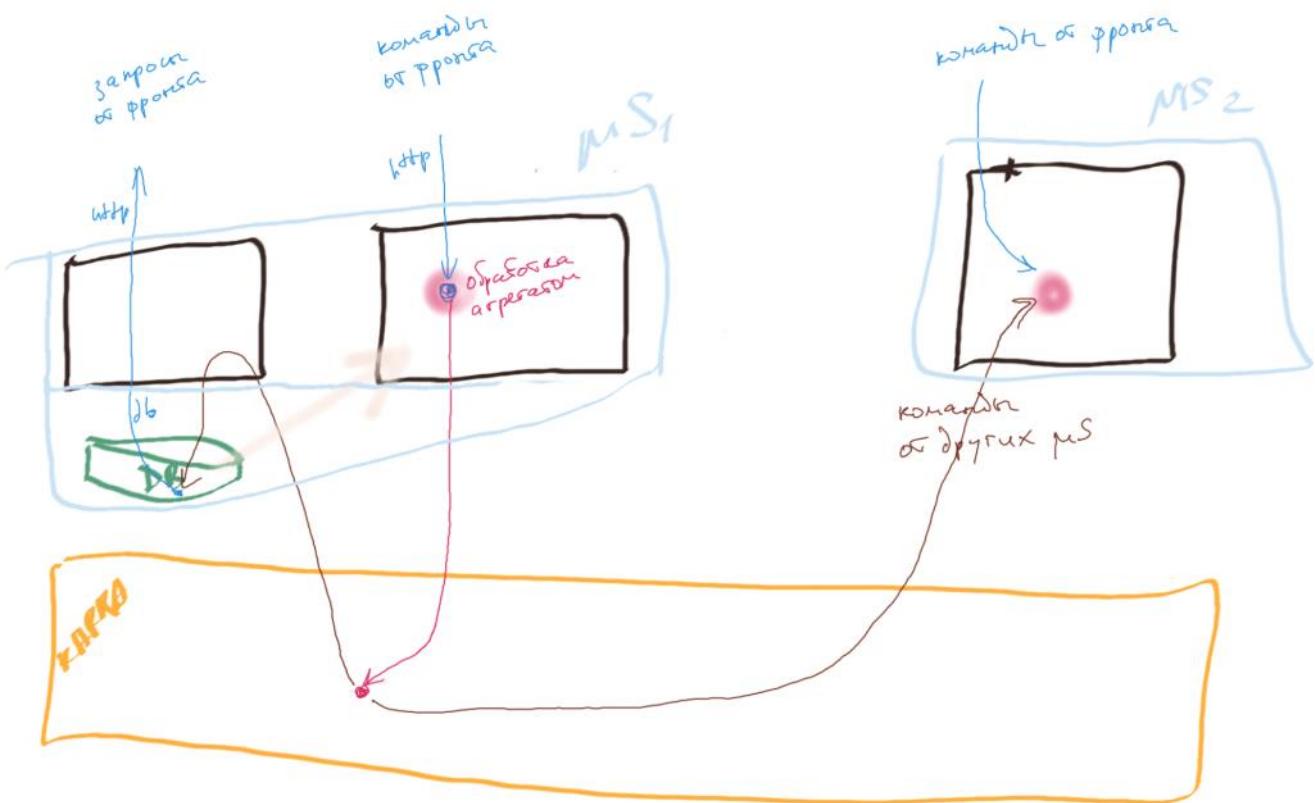
вариант1) микросервис работает только с событиями из кафки,
и после обработки также кладет событие обратно в кафку (откуда оно также может забраться этим же микросервисом для своего query store)



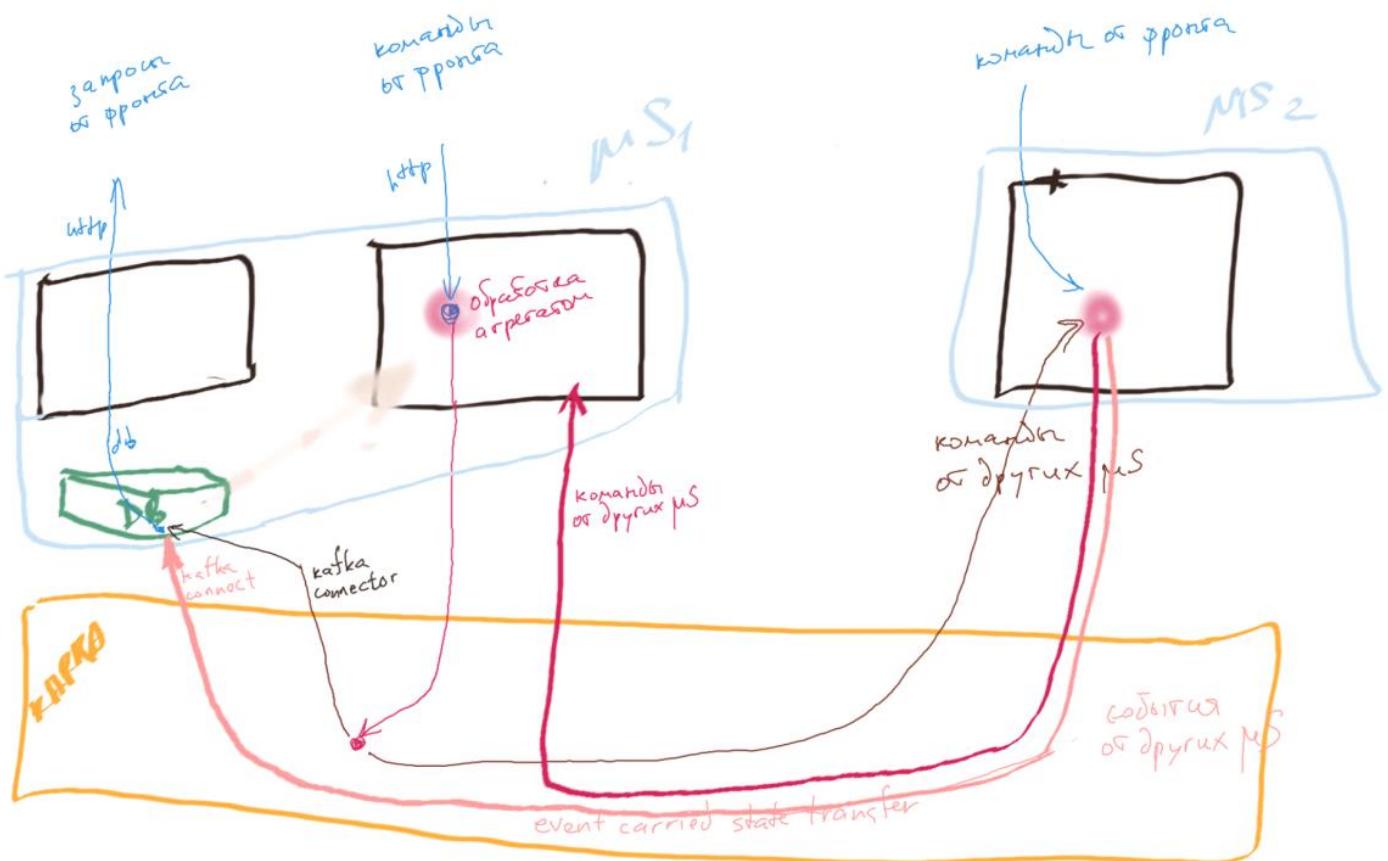
вариант2)



вариант3)

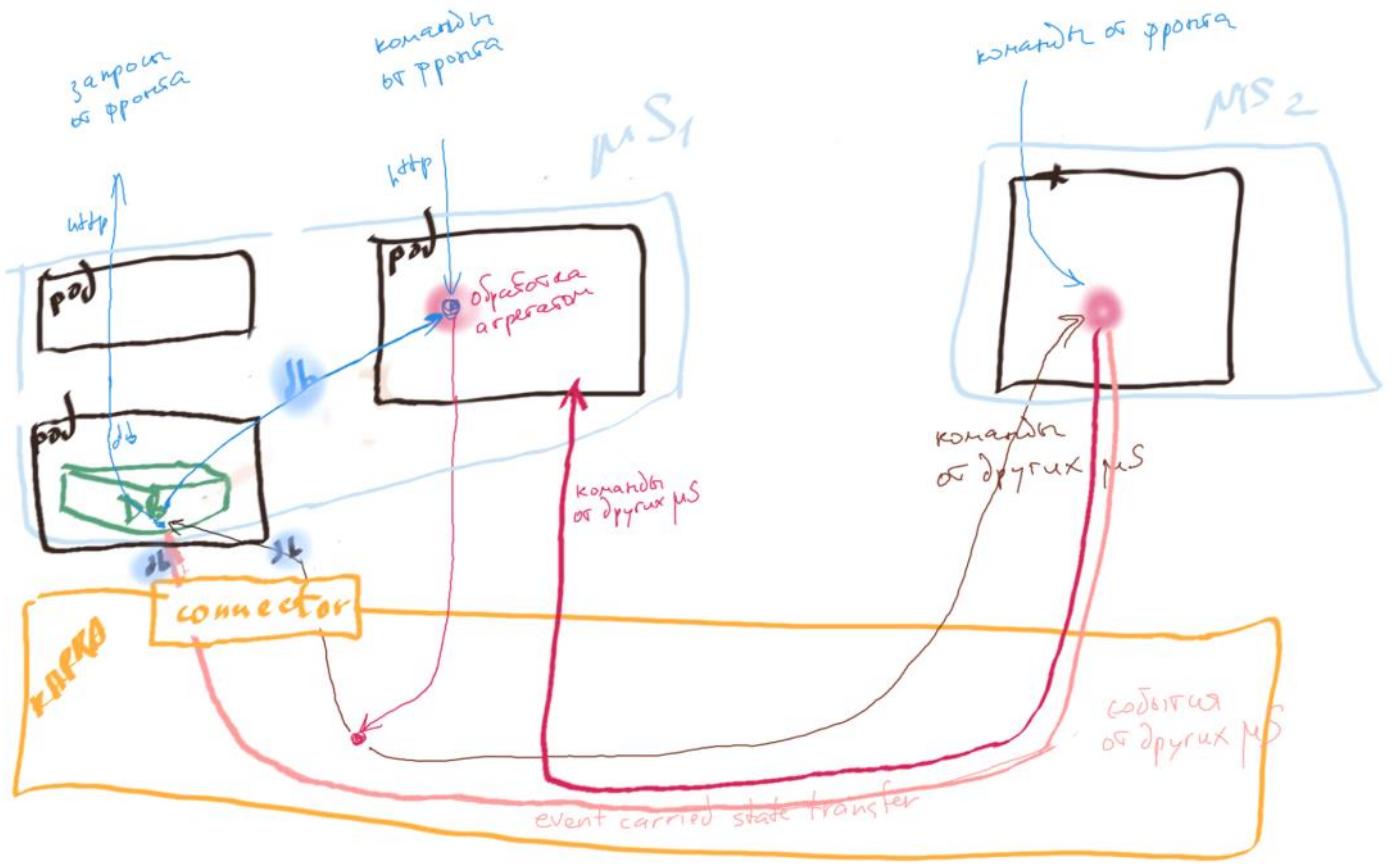


вариант 4

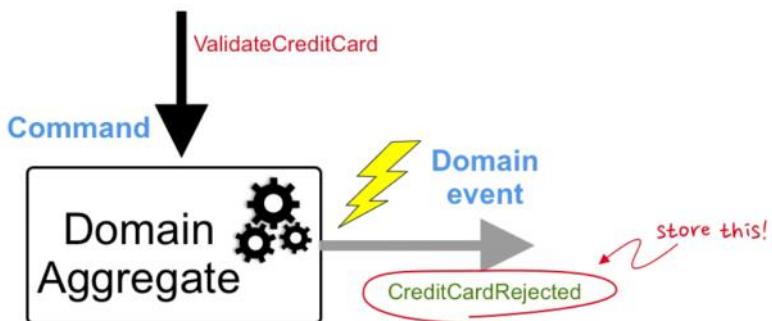


- но тогда возникают вопросы: как автоматически восстановить БД после сбоя из топика



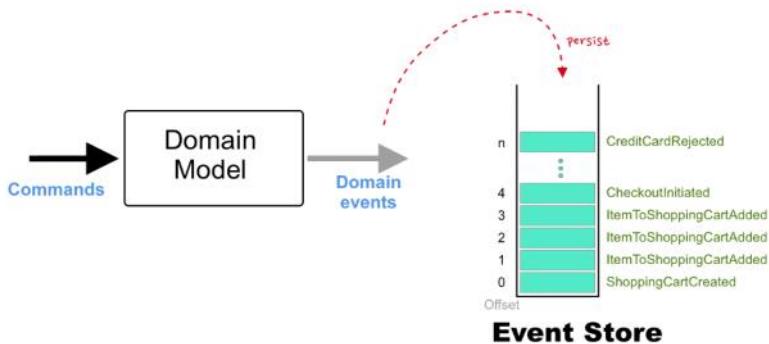


Event Sourcing



- Instead of only working with the state of the world, as it is "now", we want to see how the world changes.
- We introduce the concept of an **event**
- Events** tell us what changed
- A **command** is applied to a **domain aggregate** (or domain object)
- A **command**, if successfully applied to the **domain aggregate** always triggers a corresponding **event**
- Let's look at a few samples:
 - AddItemToShoppingCart → ItemToShoppingCartAdded
 - CheckoutShoppingCart → CheckoutInitiated
 - ValidateCreditCard → CreditCardValidated or CreditCardRejected
- Instead of updating the current state (of the domain aggregate) in a database we store the events in a (commit) log

The Event Store

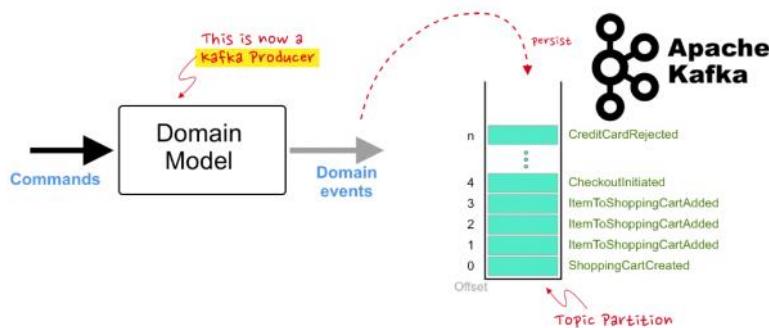


- Events are stored in a structure often called a **commit log**
- A new event is appended **at the end** of the log
- Existing events are immutable, that is, they are never updated or deleted
- The effect of wrong events must be corrected by so called **compensating events**
- The **current state** of the world can be achieved by replaying all events
- We can get the **state at any time** in the past by replaying the events up to that particular point in time



Broken down to the Kafka world: All these events got to the same topic and the key is e.g. `shoppingCartId` such as that all events belonging to the same cart go to the same partition and thus retain ordering.

Kafka as Event Store



Of course, when observing the previous slide with the **event store** we should immediately feel familiar. Doesn't this resemble to what Kafka offers us in the first place? And indeed, Apache Kafka is the ideal event store with all the guarantees we expect:

- High volume
- Durability
- Availability
- Append only commit log

Perfect! Here we thus see the connection between the generic event driven architecture and the Confluent Platform driven by Apache Kafka.

Promise Theory

9 января 2021 г. 15:32

Complexity Theory and Promise Theory and Theory of Constraints,

- это набор принципов, основанных на формальных рассуждениях, без чрезмерно ограниченных идей логики. Некоторые люди (не математики) могут назвать это математическим, потому что в нем есть формальные подходы, основанные на правилах или ограничениях
- Слово обещание показалось мне подходящим для того, что мне было нужно: своего рода атом для намерения, который, в сочетании, мог представлять поддерживаемую политику. Однако быстро стало ясно (открыв ящик Пандоры по поводу этой идеи), что происходит что-то более общее, что необходимо понять в отношении обещаний. Обещания также могут быть эффективным способом понимания целого ряда связанных вопросов, связанных с теми же самими работают в целом, и это обещало кое-что, что ранее не воспринималось всерьез: способ объединить поведение человека и машины в одном описании
- Первое, что мы замечаем, это то, что некий агент (человек или робот) должен дать обещание, поэтому мы знаем, кто является активным агентом, и что, давая обещание, этот агент принимает на себя ответственность за него. Второе, что мы замечаем, - это отсутствие мотивации давать обещания. Сотрудничество обычно предполагает диалог и мотивацию. В этих обещаниях отсутствует аналог, например: Я обещаю заплатить вам, если чаши будут чистыми. Таким образом, перспектива обещания приводит к вопросу: как мы документируем стимулы?

требование

Обещание выражает намерение относительно конечной точки или конечного результата, вместо того, чтобы указывать, что делать в начальной точке.

- Команды делаются в зависимости от того, где вы находитесь
- Обещание конечного состояния не зависит от текущего состояния дел (см. Рис. 1-1).
- Обещания также выражают намерение с точки зрения максимальной уверенности. Обещания относятся к вам (самому себе) - агенту, который их делает. По определению автономии каждый агент гарантированно может контролировать это «я».
- Навязки или команды - это то, что применимо к другим (не к себе). Это, по определению, то, что вы не контролируете.
- Можно пообещать что-то относительно отправной точки: «Я обещаю встать на голову прямо сейчас». «Я выйду из дома в 9:00». Но обещания представляют собой постоянные, постоянные состояния, в которых команды не могут. Они описывают преемственность.
- Требование - это обязательство от места обобщения высокого уровня до места более экспертного исполнения.

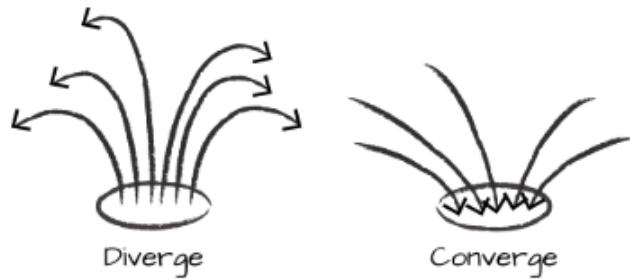


Figure 1-1. A theme we'll revisit several times in the book is the divergence or convergence of timelines. On the left, commands fan out into unpredictable outcomes from definite beginnings, and we go from a certain state to an uncertain one. On the right, promises converge towards a definite outcome from unpredictable beginnings, leading to improved certainty.

Не говорите мне, что вы делаете, скажите мне, что ты пытаешься достичь!

- Не говорите мне, что вы делаете, скажите мне, что ты пытаешься достичь! » То, что вы на самом деле делаете, может вообще не иметь отношения к тому, чего вы пытаетесь достичь.

Требования вводятся сверху вниз.

- Теория обещаний сосредотачивает внимание на активных агентах по двум причинам: во-первых, потому, что они больше всего знают о своей способности выполнять обещания. Во-вторых, потому что активные агенты - это атомные строительные блоки, которые можно легко добавить в любой более крупный акт сотрудничества.
- Бросить кому-нибудь мяч без предупреждения - это навязывание. Не было заранее запланированного обещания, которое заранее объявляло о намерении; мяч просто прицелили и бросили. Определение, очевидно, не подразумевает обязанности поймать мяч. Наложенный (ловец) может быть не в состоянии принять наложение или может не желать его принимать. В любом случае, автономия ловца не нарушается тем фактом, что бросающий пытается навязать ему.

Некоторым людям теория обещаний кажется перевернутой. Они хотят мыслить категориями обязательств.

Наложения и обязательства - это рецепт конфликта

- потому что источник намерений распределен между несколькими агентами. Обещания можно давать только о себе, поэтому они всегда носят локальный и добровольный характер. В навязывании можно отказаться, но это также может привести к путанице.

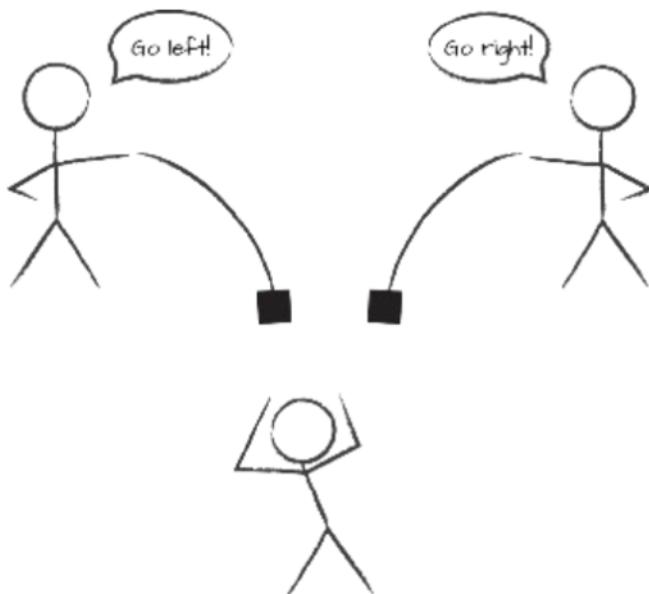


Рисунок 5-11. Конфликты возникают сразу же из-за навязывания, потому что намерение не является локальным и добровольным.

намерение - Простое выражение намерения - это то, что мы называем обещанием .

- **Намерение** Это предмет какого-то возможного исхода. Это то, что можно интерпретировать как имеющее значение в конкретном контексте. Любой агент (человек, объект или машина) может таить намерения. Намерение может быть чем-то вроде «быть красивым» для света или «выиграть гонку» для спортсмена.
- **Обещание** Когда намерение публично объявляется аудитории (так называемая сфера действия), оно становится обещанием. Таким образом, обещание - это заявленное намерение. В этой книге я буду говорить только о так называемых обещаниях первого рода, что означает обещания о самом себе. Другими словами, мы устанавливаем правило: ни один агент не может давать обещание от имени другого
- **Обязательство** Наложение штрафных санкций за несоблюдение. Это более агрессивно, чем простое навязывание
- **Оценка** Решение о том, выполнено ли обещание. Каждый агент дает свою оценку обещаниям, которые ему известны. Часто оценка включает наблюдение за поведением других агентов.
- Теория обещаний - это, если хотите, подход к моделированию кооперативных систем, который позволяет вам спросить: «Насколько мы уверены, что это сработает?» и «С какой скоростью?» Вы можете начать отвечать на такие вопросы только в том случае, если агентство, участвующее в возможном совместном обещании «заставить это работать», может обещать некоторые базовые предпосылки.
- Намерение - это то неуловимое качество, которое описывает цель. Это отчетливо человеческое суждение. Когда мы что-то намереваемся, это придает смысл нашим великим замыслам.

Intention

This is the subject of some kind of possible outcome. It is something that can be interpreted to have significance in a particular context. Any agent (person, object, or machine) can harbour intentions. An intention might be something like “be red” for a light, or “win the race” for a sports person.

Promise

When an intention is publicly declared to an audience (called its *scope*) it then becomes a promise. Thus, a promise is a stated intention. In this book, I'll only talk about what are called promises of the first kind, which means promises about oneself. Another way of saying this is that we make a rule: no agent may make a promise on behalf of any other (see [Figure 1-2](#)).

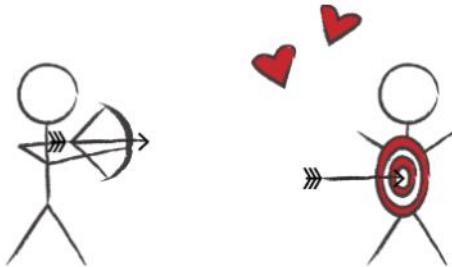


Figure 1-2. A promise to give is drawn like Cupid's arrow...

Imposition

This is an attempt to induce cooperation in another agent (i.e., to implant an intention). It is complementary to the idea of a promise. Degrees of imposition include hints, advice, suggestions, requests, commands, and so on (see [Figure 1-3](#)).

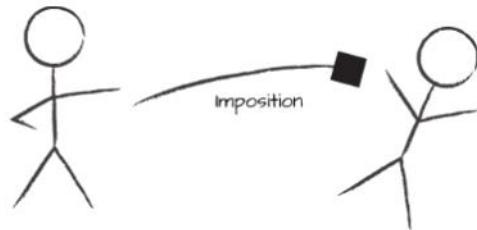


Figure 1-3. An imposition is drawn like a fist.

Obligation

An imposition that implies a cost or penalty for noncompliance. It is more aggressive than a mere imposition.

Assessment

A decision about whether a promise has been kept or not. Every agent makes its own assessment about promises it is aware of. Often, assessment involves the observation of other agents' behaviours.

конфликтом намерений

- Если бы все агенты разделяли одни и те же намерения, в обещаниях не было бы особой нужды. Все ладили и пели в полной гармонии, работая для достижения общей цели. Тот факт, что начальное состояние системы имеет неизвестные намерения и распределенную информацию, означает, что мы должны установить такие вещи, как соглашения, в которых агенты обещают вести себя определенным образом. Это то, что мы называем оркестровкой.

обещание

Эта идея, что каждый автономный агент имеет собственное независимое представление, означает, что агенты также независимо формируют ожидания.

Эта идея, что каждый автономный агент имеет собственное независимое представление, означает, что агенты также независимо формируют ожидания. Это, в свою очередь, позволяет им выносить суждения, не дожинаясь проверки результатов. Вот как мы используем обещания в тандеме с доверием. Каждый возможный наблюдатель, имеющий доступ к части информации, может индивидуально сделать оценку и, учитывая свои различные обстоятельства, может прийти к разным выводам.

Это более разумная версия банального бизнес-афоризма о том, что «покупатель всегда прав». Каждый наблюдатель имеет право на свою точку зрения. Полезный побочный эффект обещаний состоит в том, что они приводят к процессу документирования условий, при которых агенты принимают решения для дальнейшего использования.

- Из этих примеров мы видим, что суть обещаний довольно общая. Действительно, такие обещания окружают нас повсюду в повседневной жизни, как в повседневной одежде, так и в технических дисциплинах. Заявления о технических спецификациях также можно рассматривать как обещания, даже если мы обычно не думаем о них таким образом.
- В действительности, обещания и навязывания всегда рассматриваются с субъективной точки зрения одного из автономных агентов. Не существует автоматически никакого «взгляда глазами Бога» на то, что знают все агенты. Мы можем назвать такой субъективный взгляд «миром» агента. В информатике мы бы назвали это распределенной системой. Обещания имеют две полярности по отношению к этому миру: внутреннюю или внешнюю от агента.
 - Обещания и навязывание чего-либо (исходящие от агента)
 - Обещания и навязывания получить что-то (внутри агента)

можно обещать только то что ты реально можешь

- Обещания выражают результаты, надеясь, в ясной форме. Если обещания неясны, они не имеют особой ценности. Успех обещаний заключается в способности утверждать о намерении вещи или группы вещей.
- В мире без доверия обещания были бы совершенно неэффективными.
- Причина, по которой мы вообще даем обещания, заключается в том, что они предсказывают благоприятные результаты в глазах некоторых наблюдателей. Они предлагают информацию, которая может позволить тем, кто знает об обещании, подготовиться и получить преимущество. Физический закон допускает инженерию, последовательные реакции организма позволяют медицину, программное обеспечение обещает целостность данных и так далее.
- Как мы можем быть уверены в том, что обещают? Раньше считалось обязательным документирование намерений, но потом мы обнаружили, что обещания можно давать, обращаясь к интуиции. Само по себе это не часть теории обещаний, но относится к тому, насколько четко мы должны сообщать о намерениях и будут ли все в системе обещаний давать одинаковые оценки. Рыночный брендинг - это пример обещаний, выдвигаемых ассоциацией.
- Согласие в группе называется консенсусом. Для того, чтобы возникло совместное поведение, не обязательно должен быть консенсус, но должно быть достаточное взаимное понимание значения обещаний.
- То, что имеет смысл обещать, - это вещи, которые, как мы знаем, можно сохранить (т. е. Привести в определенное состояние и поддерживать).
 - Состояния, расположения или конфигурации, например макет
 - Идемпотентные операции (вещи, которые происходят один раз): удалить файл, очистить корзину
 - Регулярное, установившееся состояние или непрерывное изменение: постоянная скорость
 - Событие, которое уже произошло

обещание можно дать только от своего лица

- Однако агенты сдерживают обещания только о собственном поведении. Если мы попытаемся дать обещания от имени других, они, скорее всего, будут отвергнуты, их невозможно будет выполнить, или другой агент может даже не знать о них.
- Теория обещаний предполагает, что агент может давать обещания только относительно своего собственного поведения (потому что это все, что он контролирует), и это решает проблемы, связанные с распространением информации, гарантируя, что оба информации и ресурсы, необходимые для решения любой проблемы, являются локальными и доступны агенту. Таким образом, агент может автономно исправить обещание, взяв на себя ответственность. В этом смысле свободы воли.

как начать обдумывать обещания: Суть в том, что обещания могут или не могут быть выполнены (по сотне разных причин).

- Обещания также заставляют задуматься о планах на случай непредвиденных обстоятельств. Что, если ваши первые предположения не оправдаются?

Identify the key players (agents of intent)

The first step in modelling is to identify the agencies that play roles within the scope of the problem you are addressing. An agent is any part of a system that can intend or promise something independently, even by proxy. Some agents will be people, others will be computers, policy documents, and so on—anything that can document intent regardless of its original source.

Table 2-1. The lifecycle of promises

Promise State	Description	not to be
proposed	A promise statement has been written down but not yet made.	blunt
issued	A promise has been published to everyone in its scope.	en an
noticed	A published promise is noticed by an external agent.	play a
unknown	The promise has been published, but its outcome is unknown.	ans in
kept	The promise is assessed to have been kept.	have a
not kept	The promise is assessed to have been not kept.	his is
broken	The promise is assessed to have been broken.	rently
withdrawn	The promise is withdrawn by the promiser.	official
expired	The promise has passed some expiry date.	com-
invalidated	The original assumptions allowing the promise to be kept have been invalidated by something beyond the promiser's control.	re real
end	The time at which a promise ceases to exist.	ealing s and ave to

work with. There are techniques for this.

The bottom line is that promises might or might not be kept (for a hundred different reasons). After all, they are only intentions, not irresistible forces.

жизненный цикл обещания

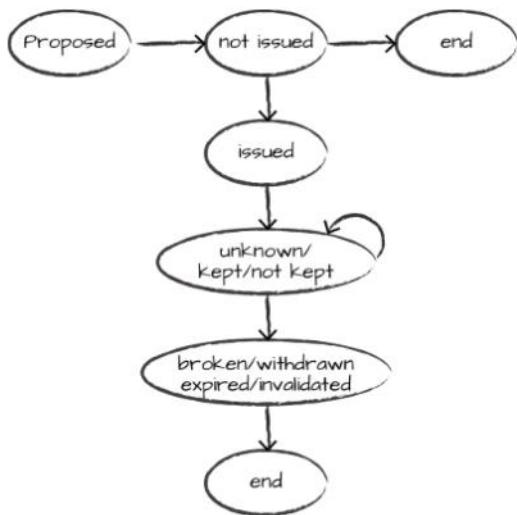


Figure 2-2. The promise lifecycle for a promiser.

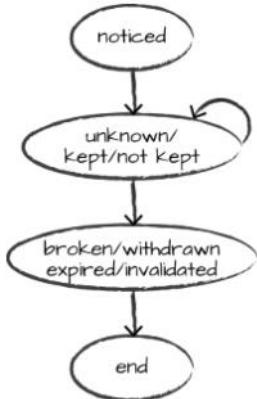


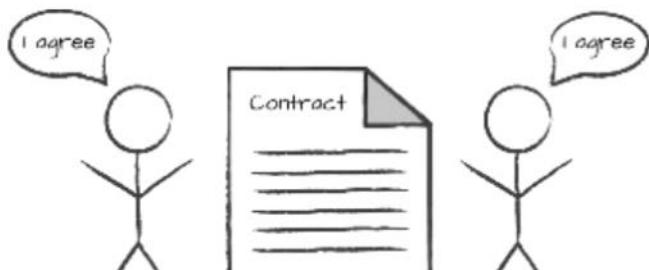
Figure 2-3. The promise lifecycle for a promisee.

обещания не должны противоречить друг другу

- Чтобы обещания приводили к определенности, они должны быть непротиворечивыми (см. Главу 6). Обещания могут существовать до тех пор, пока они не накладываются друг на друга и не наступают на пятки другим. Избежание конфликта между агентами приводит к определенности в группе. В конечном итоге теория обещаний упрощает разрешение конфликтов.

Если агенты обещают принять предложение от другого, то они подтверждают свою веру в него.

- Предположим, у нас есть один или несколько агентов, обещающих принять обещание. Наблюдатель, увидев такое обещание принять предложение, мог заключить, что все агенты (обещавшие принять одно и то же предложение) на самом деле согласились с их позицией по этому вопросу. Таким образом, если бы было три агента и только двое согласились, это выглядело бы, как на рисунке 5-4 .
- Например, когда клиент подписывает договор страхования, он соглашается с предлагаемым содержанием, и договор вступает в силу с момента подписания. А как насчет подписи страховой компанией? Иногда это явно, иногда неявно.



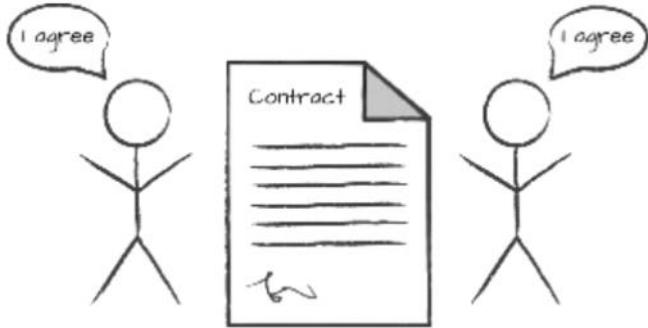


Рисунок 5-7. Подпись - это обещание принять предложение.

баллистические рассуждения

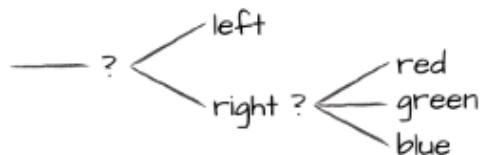


Figure 4-4. Decisions lead to many possible worlds or outcomes, like a game of billiards. This is “ballistic reasoning” or deterministic reasoning.

Теория игр, на которую опирается Теория обещаний,

- долгосрочный успех в торговле обещаниями может быть ускорен краткосрочной готовностью пойти на компромисс в отношении собственных эгоистических потребностей (т. Е. Принять предполагаемые недостатки другого агента, не отказываясь взамен от собственных услуг).
- Когда один агент дает обещание предложить свою услугу (+), а другой принимает (-) это обещание, это позволяет первому агенту передать информацию второму. Оба обещания должны быть повторены в обратном порядке, иначе второй агент сможет передать информацию обратно первому.

Один из важных уроков теории обещаний состоит в том, что семантика и намерение всегда подчинены динамическим возможностям ситуации.

- Если что-то не может быть реализовано на самом деле, это не помогает яростно задуматься. Вода не будет низко подниматься в гору, каковы бы ни были намерения. С этими предостережениями можно попытаться распространить новые действенные намерения в группе, начав с чего-то и дождавшись, пока изменения уравновесятся через уже существующие каналы сотрудничества. Чем лучше мы понимаем лежащие в основе обещания, тем легче будет взломать код.

eventually consistency

- Координационные агентства в распределенных системах путем обмена информацией являются основой многих технологий. Программные системы, такие как CFEengine,

Kubernetes, Zookeeper и т. д., Используют такие алгоритмы, как Paxos, Raft и их варианты, чтобы гарантировать согласованность данных, совместно используемых агентами. Стоимость обещания распределенной согласованности может быть высокой в зависимости от того, что именно обещано. Термин возможная согласованность часто используется там, где агентам в системе не требуется общее чувство времени. Во всех случаях Promise Theory сообщает нам, что существует два подхода к уравновешиванию данных: одноранговый подход или централизованный ведущий подход. В большинстве случаев используется центральный главный подход, а главные агенты резервного копирования обмениваются данными как одноранговые узлы, создавая гибридное решение. Централизованный подход ограничен вертикальным масштабированием, тогда как одноранговый подход может масштабироваться горизонтально.

- Поскольку последовательность и ожидания важны, например, для репутации, многие предприятия, поставщики товаров и услуг хотели бы иметь возможность обещать согласованность во всех своих точках обслуживания, но к настоящему времени должно быть ясно, что это не так. имеет смысл.
- Для этого временного интервала потребуется достаточный резерв, чтобы информация могла уравновеситься в системе (т. Е. Перемещаться от одного конца системы к другому и обратно). Банки сознательно используют этот подход, когда прекращают торговлю в конце каждого дня и заставляют клиентов ждать, скажем, три рабочих дня для проверки транзакции, несмотря на современные средства связи. В некоторых обстоятельствах можно поставить этот детерминизм выше точности определения времени, но здесь тоже есть компромисс. Как бы мы ни выбирали определение согласованности, нам приходится иметь дело с искажением истории данных, искусственным снижением производительности захвата или и тем, и другим.
- Нам нужно привыкнуть к идее множества миров и только локальной согласованности, когда мы направляемся в будущее информационных систем. По мере того, как расстояния становятся больше, а временные рамки становятся короче, способность обещать консенсус между агентами теряет смысл, и нам лучше справляться с несоответствиями, чем пытаться заставить их исчезнуть.
- Если бы сеть супермаркетов пообещала, что во всех своих магазинах будет полностью постоянный запас, вы бы поверили этому обещанию? Почему или почему нет? Если банк обещает, что баланс вашего счета будет известен всем его филиалам по всему миру, поверите ли вы этому обещанию? Если ресторан обещает, что все участники званого ужина получат одинаковую еду, доверяете ли вы этому?

принцип неизменности намерений

- Дело в том, что люди - существа привычки или постоянной семантики. Изменения в семантике приводят к переоценке доверительных отношений, будь то между людьми напрямую или по доверенности через технологии.
- Способ избежать этого - придерживаться принципа неизменности намерений. На самом деле это ключ к философии, которая выросла вокруг так называемого непрерывного предоставления услуг (см. Главу 8). Можно обещать изменения постоянно, пока основная семантика остается неизменной.
- Непрерывная доставка - это то, что вы получаете, когда находитесь в самолете. Самолет не должен упасть с неба во время оказания услуги. Так же, как и непрерывность исполнения, существует преемственность цели. Самолет должен лететь в один пункт назначения и не постоянно менять свое мнение. Обещания должны длиться ровно столько, сколько нужно, чтобы их сдерживать, иначе они станут обманом

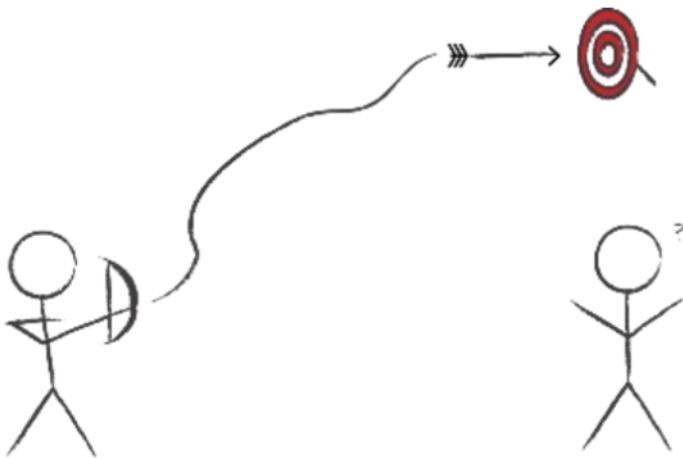


Рисунок 5-15. Приближение к цели обычно означает предсказуемое поведение! Отсутствие постоянных обещаний может привести к восприятию беспорядочного поведения.

количество обещаний, которые мы можем выполнить, ограничено

- Более того, чем ближе или интимнее мы пересматриваем наши отношения, тем меньше мы можем поддерживать, потому что поддерживать близкие отношения стоит дороже. Это должно сыграть роль в экономике сотрудничества. Другими словами, отношения стоят дорого, а уровень внимания ограничивает то, сколько мы можем поддерживать. Это означает, что количество обещаний, которые мы можем выполнить, ограничено. У каждого агента есть бесконечные ресурсы.

карточный домик

Например, если агент обещает использовать службу X, чтобы сдержать свое собственное обещание Y, то мы можем сделать вывод, что обещание Y неустойчиво к потере службы X. Это цепочки обязательной зависимости (см. Рисунок 6- 1).

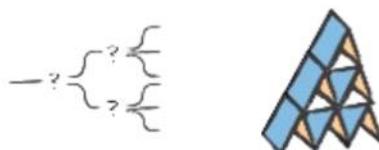


Рисунок 6-1. Рассуждения приводят к ветвлению процессов, которые расширяют множество возможностей и уменьшают уверенность. Результат может быть хрупким, как карточный домик.

сеть обещаний

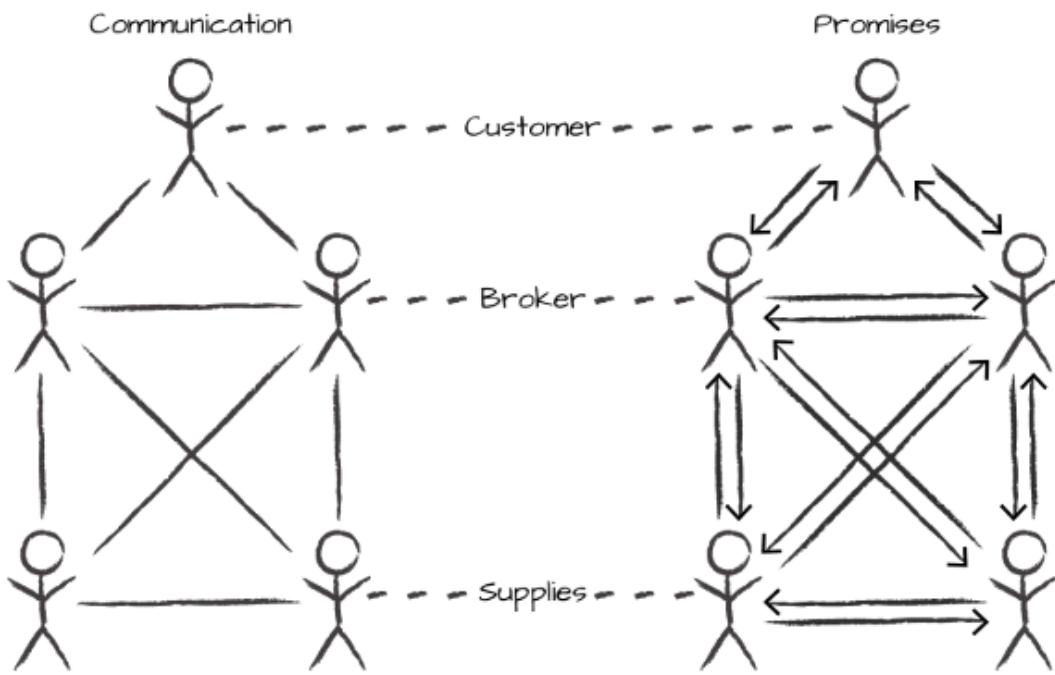


Figure 5-2. A stakeholder, wanting to use a promise provided redundantly, has to know all the possible agencies and their backups making the promise. It also wants a promise that all agents are equivalent, which means that those agencies need to promise to collaborate on consistency. These agents in turn might rely on other back-office services, provided redundantly, and so on, all the way down. Vertical promises are delegation promises, and horizontal promises are collaboration/coordination promises.

Этот процесс представляет собой способ решения теоретико-игрового решения с помощью равновесия по Нэшу

- Урок состоит в том, что в мире неполной информации доверие может помочь вам или навредить вам, но это игра. Задача информатики состоит в том, чтобы определить согласие, основанное только на информации, которую агент может обещать самому себе.
- Аксельрод показал, что равновесие по Нэшу из теории экономических игр позволяет сотрудничеству происходить достаточно автономно, целиком и полностью основанным на индивидуальных обещаниях и оценках. Он не использовал язык обещаний, но намерение было тем же. Агенты могли воспринимать ценность взаимных обещаний совершенно асимметрично, если они проявляли последовательность в своих обещаниях

информацией о подходящем времени перекрытия. Этот процесс представляет собой способ решения теоретико-игрового решения с помощью равновесия по Нэшу.

Имея только частичную информацию, прогресс все же может быть достигнут, если агенты доверяют друг другу информацию об истории предыдущих коммуникаций.

1. А обещает встретиться в среду или пятницу с В, С и D.
2. D и В обещают друг другу, что они оба могут встретиться во вторник или пятницу, учитывая знание пункта 1.
3. D и С обещают друг другу встретиться во вторник или четверг, если они знают пункты 1 и 2.
4. А осведомлен об обещаниях В и С, но не может связаться с D.

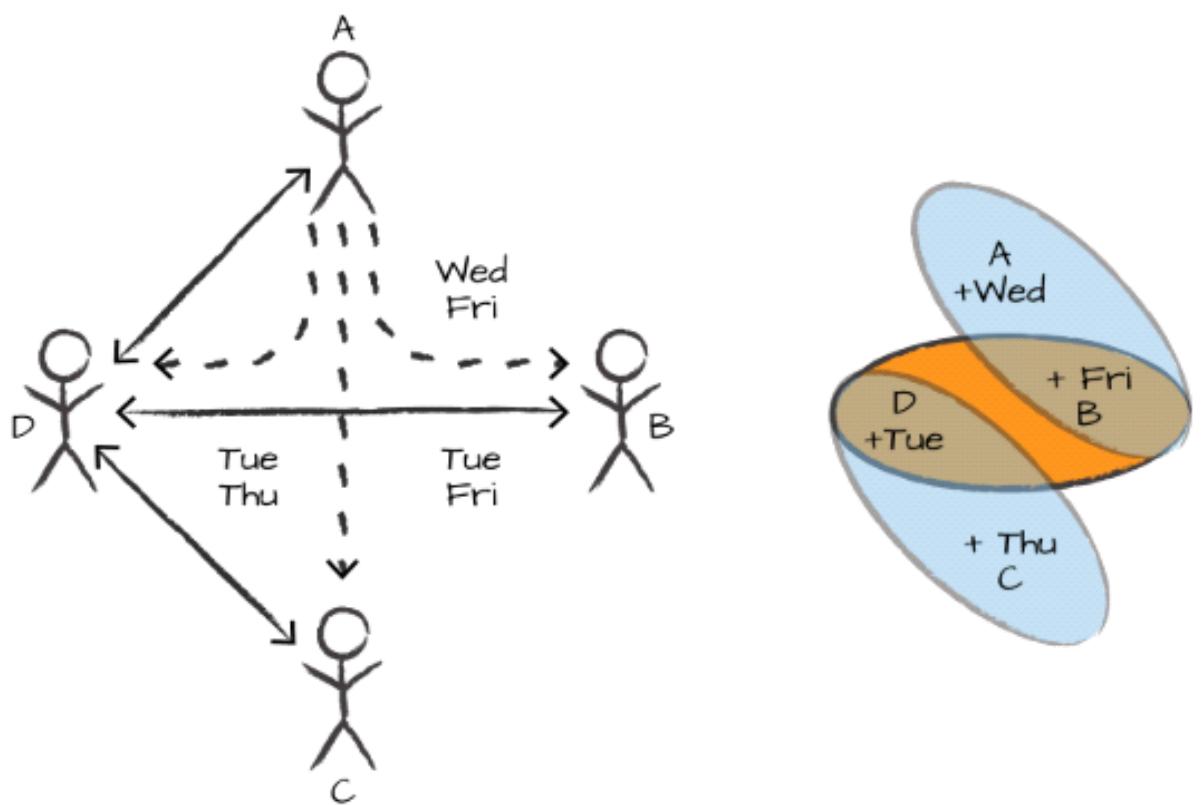


Figure 5-8. Looking for a Nash equilibrium for group agreement.

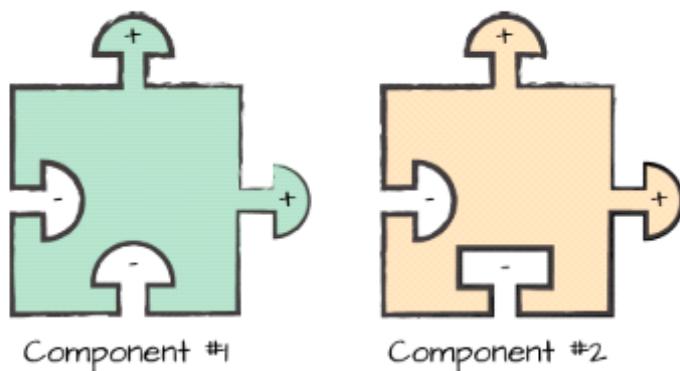


Figure 6-3. Promises that give something (+), and promises that accept something (-), match in order to form a binding, much like a jigsaw puzzle.

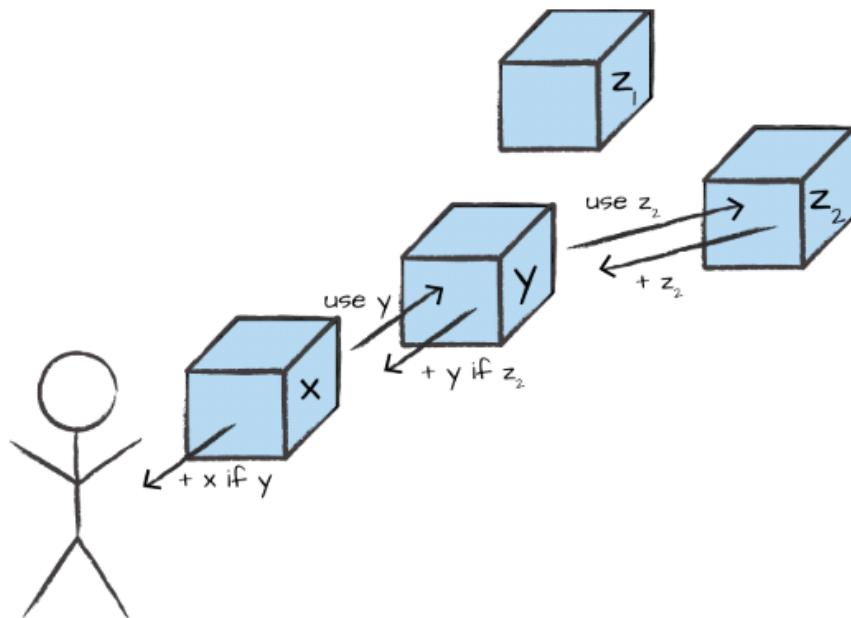


Figure 6-9. A hierarchy of component dependencies.

как рассматривать системы с точки зрения обещаний

в децентрализованной системе все медленнее (чем в централизованной)

- Преимущество децентрализации заключается в том, что меньше узких мест, ограничивающих рост системы. Недостатком является то, что агенты не подчиняются однозначно одному агентству, поэтому контроль труднее воспринимать и медленнее уравновешивать. Таким образом, децентрализованные системы демонстрируют масштабируемость без грубой силы за счет того, что их труднее понять и они медленнее изменяются.

Компонент - это объект, который ведет себя как агент и дает ряд обещаний другим агентам в контексте системы.

- Его можно представить в виде связи обещаний, исходящих из определенного места. Компоненты играют роль в рамках более крупной системы. Для слабосвязанных систем компоненты можно назвать автономными.
Когда объект является компонентом, он отделяется от источника своего намерения. Это еще один кирпич в стене .

Могут ли сами агенты иметь компоненты? (Superagents)

larger system. Typical promises that components would be designed to keep could include:

- To be replaceable or upgradable
- To be reusable, interchangeable, or compatible
- To behave predictably
- To be (maximally) self-contained, or avoid depending on other components to keep their promises

роль компонента

- Одна из простых структур для шаблонов обещаний - это сопоставление компонентов с ролями. Компонент - это просто набор из одного или нескольких агентов, которые формируют роль посредством ассоциации или сотрудничества. После включения в более крупную систему эти роли также могут быть назначены через связывающие отношения с пользователями и поставщиками. Таблица 6-1 иллюстрирует некоторые примеры компонентов и их роли в некоторых общих системах.

Table 6-1. Examples of system components

System	Component	Role
Television	Integrated circuit	Amplifier
Pharmacy	Pill	Sedative
Patient	Intravenous drug	Antibiotic
Vertebrate	Bone	Skeleton
Cart	Wheel	Mobility enabler
Playlist	Song by artist	Soft jazz interlude

Регрессионный тест включает оценку того, выполняются ли обещания компонентов после изменения компонента.

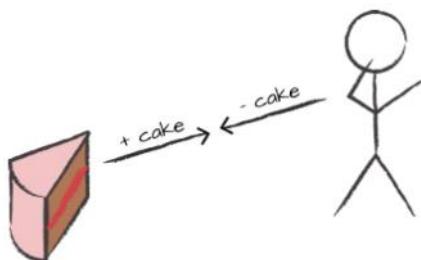


Figure 6-7. A regression test involves assessing whether component promises are still kept after changing the component. Does today's piece of cake taste as good as yesterday's piece?

обещания только для конкретных версий компонентов

- Если компоненты обещают использование зависимостей в зависимости от версии, поддерживающие компоненты могут сосуществовать без конфликта. Обещания одного компонента использовать другой можно избежать, если обратиться к конкретным версиям, которые содержат их обещание использовать.

Это позволяет избежать риска последующего аннулирования обещаний компонентом по мере развития версий зависимостей.

брендинг - Компоненты должны каким-то образом обещать свою функцию, с некоторым представлением, которое узнают пользователи.

Какие атрибуты квалифицируются как имена? Имена не обязательно должны быть словами. Имя могло быть формой. Чаще не нужно имя, чтобы мы ее узнали. Компоненты должны каким-то образом обещать свою функцию, с некоторым представлением, которое узнают пользователи. Именование в самом широком смысле позволяет компонентам быть узнаваемыми и различимыми для повторного использования. Узнавание может происходить через форму, текстуру или любую форму чувственной передачи сигналов.

В маркетинге такой вид нейминга часто называют брендингом. Концепция брендинга восходит к обозначению следов, нанесенных на домашний скот. В маркетинге брендинг рассматривает многие аспекты психологии наименования, а также то, как привлечь внимание зрителей, чтобы сделать одни компоненты более привлекательными для пользователей, чем другие.

Имеет смысл, что пакеты обещаний заключают соглашения об именах в системе в соответствии с их функциональной ролью, позволяя компонентам:

- Быстро узнаваемый для использования
 - Отлично от других компонентов
 - Создан на основе культурных или инстинктивных знаний
- Поскольку агент не может навязывать свое имя другому агенту и не может знать, как другой агент относится к нему, соглашения об именах должны быть общеизвестными в рамках совместной подсистемы.
Установление имен и доверие к ним - одна из фундаментальных проблем в системах агентов.
- Именование должно отражать изменения во времени и пространстве, используя номера версий и временные метки или другие виртуальные координаты местоположения

Теория обещаний делает простые прогнозы относительно услуг, которые, возможно, противоречат здравому смыслу

- Это говорит нам о том, что ответственность за получение услуг в конечном итоге лежит на клиенте, а не на сервере. Это проясняет несколько вещей о конструкции систем, в которых сервисы играют роль.
- Каждый потребитель или поставщик услуги остается центром своей власти и принятия решений, и обещание потребить услугу, которая еще не предоставлена, действует как неагрессивный сигнал для других агентов. добровольно предоставлять услуги, предполагая, что экономические стимулы взаимовыгодны.
- Ответственность за получение лежит на клиенте, потому что никакая услуга не может быть передана без принятия клиентом обещания услуги от сервера. Даже если есть несколько серверов, общающихся обслуживание с избыточностью, так что возможность обслуживания гарантирована, клиент должен сделать выбор в пользу использования одного или нескольких из них. Единственной точкой отказа является клиент

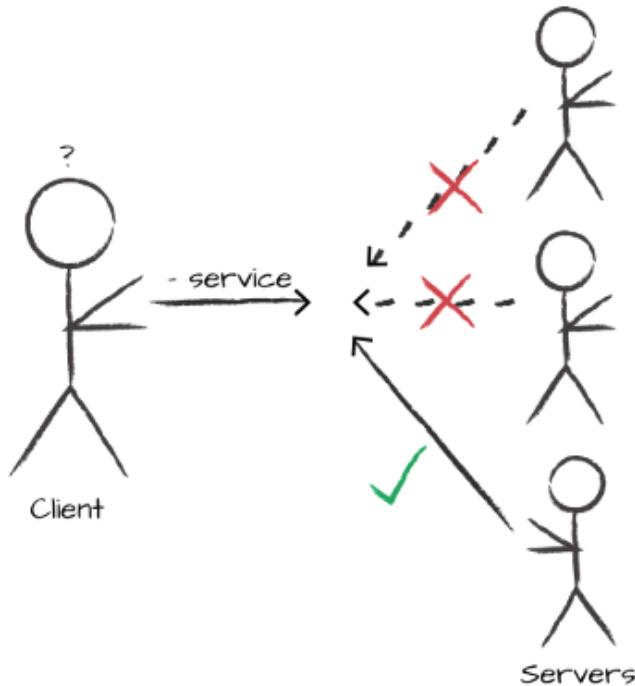


Figure 7-1. A client has to promise to use a service (-) in order to avail itself of the offer (+). No matter how many servers there are, failing or keeping their service promise, the client has to choose and accept a good one. So the client has ultimate responsibility for getting service.

Преимущества системы обычно возникают из взаимодействия между системой и пользователем, так почему мы упорствуем в этом разделении?

- Задача при проектировании системы состоит в том, чтобы включить пользователей как часть системы и относиться к обещаниям, в которых они участвуют, так же, как и к любым другим участникам механизма. Пользователь - часть машины.

Доверие - дешево, а проверка - дорого.

- Доверие - дешево, а проверка - дорого. Стоимость полностью обещанной непрерывной непрерывной доставки растет пропорционально квадрату количества посредников. Эту стоимость сквозного обеспечения можно сравнить со стоимостью подхода «гнев и забыл» с использованием только гарантий ближайшего соседа, что линейно зависит от количества агентов. Таким образом, мы видим привлекательность того, чтобы все оставалось на волю случая.

В худшем случае можно не давать никаких обещаний агентам и просто посмотреть, что произойдет

Полезность документирования и выполнения всех этих обещаний заключается в том, чтобы увидеть, насколько необходимо снизить уровень информации о намерениях агентов и где могут возникнуть потенциальные точки отказа из-за недостаточной оперативности при выполнении обещаний.

Цепочка доверия подразумевается в любом сотрудничестве, с обещаниями или без них. Если бы каждый агент не разговаривал со своими зависимостями и зависимыми объектами, агент службы не смог бы определить, искает ли какой-либо промежуточный прокси одно из обещаний. Таким образом, отсутствие доверия к доверенным лицам заставляет нас требовать больше обещаний на определенных условиях. Это увеличивает дополнительные накладные расходы и расходы.

Напомним, что мы определяем агентов как части системы, которые могут изменяться независимо

- В любое время набор обещаний может быть продублирован, как деление клеток, а затем изменен. Это механизм, используемый в информационных технологиях для разветвления процессов времени выполнения, а также разветвление проектов разработки в системе контроля версий

Ценность услуги заключается в ее простоте (т. е. в той степени, в которой она может сократить дорогостоящее взаимодействие до более простого). Чтобы потенциальные клиенты пользовались доверием, службы должны четко афишировать свои обещания, включая риски.

свойства системы

Такое негативное свойство, как хрупкость, кажется странным, но такое обещание может быть очень важным. Мы маркируем хрупкие коробки для транспортировки именно для того, чтобы убедить агентов по доставке проявить к ним особую осторожность или уменьшить нашу ответственность в случае аварии.

Continuity

The observed constancy of promise-keeping, so that any agent using a promise would assess it to be kept at any time.

Stability

The property that any small perturbations to the system (from dependencies or usage) will not cause its promises to break down catastrophically.

Resilience (opposite of fragility)

Like stability, the property that usage will not significantly affect the promises made by an agent.

Redundancy

The duplication of resources in such a way that there are no common dependencies between duplicates (i.e., so that the failure of one does not imply the failure of another).

Learning (sometimes called anti-fragility)

The property of promising to alter any promise (in detail or in number) based on information observed about an agent's environment.

Adaptability

The property of being reusable in different scenarios.

Plasticity

The property of being able to change in response to outside pressure without breaking the system's promises.

Elasticity

The ability to change in response to outside pressure and then return to the original condition without breaking the system's promises.

Scalability

The property that the outcomes promised to any agent do not depend on the total number of agents in the system.

Integrity (rigidity)

The property of being unaffected by external pressures.

Security

The promise that all risks to the system have been analyzed and approved as a matter of policy.

Ease of use

The promise that a user will not have to expend much effort or cost to use the service provided.

СЛОЖНОСТЬ = ЭНТРОПИЯ

- Сложность измеряется взрывом возможных результатов, которые развиваются по мере развития системы. Это рост информации (также называемый энтропией). Когда части системы сильно взаимодействуют, это означает, что изменение в одном месте приводит к изменению в другом. Это приводит к хрупкости и хрупкости, а не к сложности. Если бы вы могли заменить 1000 пенни банкнотой в 10 фунтов стерлингов, с этим было бы легче иметь дело.
- так называемые бифуркации или ветвления результатов, которые приводят к взрыву состояний и частей. Это вызвано сильной связью, но не ею. Действительно, путаница уменьшает общее количество результатов, так что по иронии судьбы это делает вещи менее сложными.
- Разделение проблем (то есть инстинкт наведения порядка) на самом деле является стратегией, которая вызывает раздвоения и, следовательно, распространение. Это приводит к сложности, потому что увеличивает количество вещей, которые должны взаимодействовать, что приводит к новой энтропии. Ветвление без обрезки вносит сложности. Одна из причин, по которой мы боимся сложности, заключается в том, что мы ее не понимаем.
- В противовес этой идеи антрополог Джозеф Тейнтер провел интересное исследование

обществ, уходящих в прошлое, в поисках причин, по которым они рухнули. Его вывод был примерно таким: по мере роста общества есть выгода от затрат на специализацию на разных ролях для расширения. Это потому, что люди недостаточно умны, чтобы быть экспертами во всем; у них ограниченная сила мозга и мышц. Но за это разделение приходится платить. Если нам нужна услуга от специалиста, мы должны переподключиться к нему. По мере того, как агентства разделяются, они часто формируют свои собственные частные языки, которые не уравновешены с населением в целом, поэтому также возникают издержки языкового барьера. Это приводит к снижению доверия. Затем агенты заставляют клиентов перепрыгивать через обруч, чтобы подтвердить себя. Рождаются бюрократия и организационная разобщенность! В конце концов, стоимость повторного подключения через барьеры (из-за потери доверительных отношений) превышает то, что могут себе позволить агенты, и система разваливается на части, когда пользователи начинают обходить барьеры. Разделение забот приводит к краху.

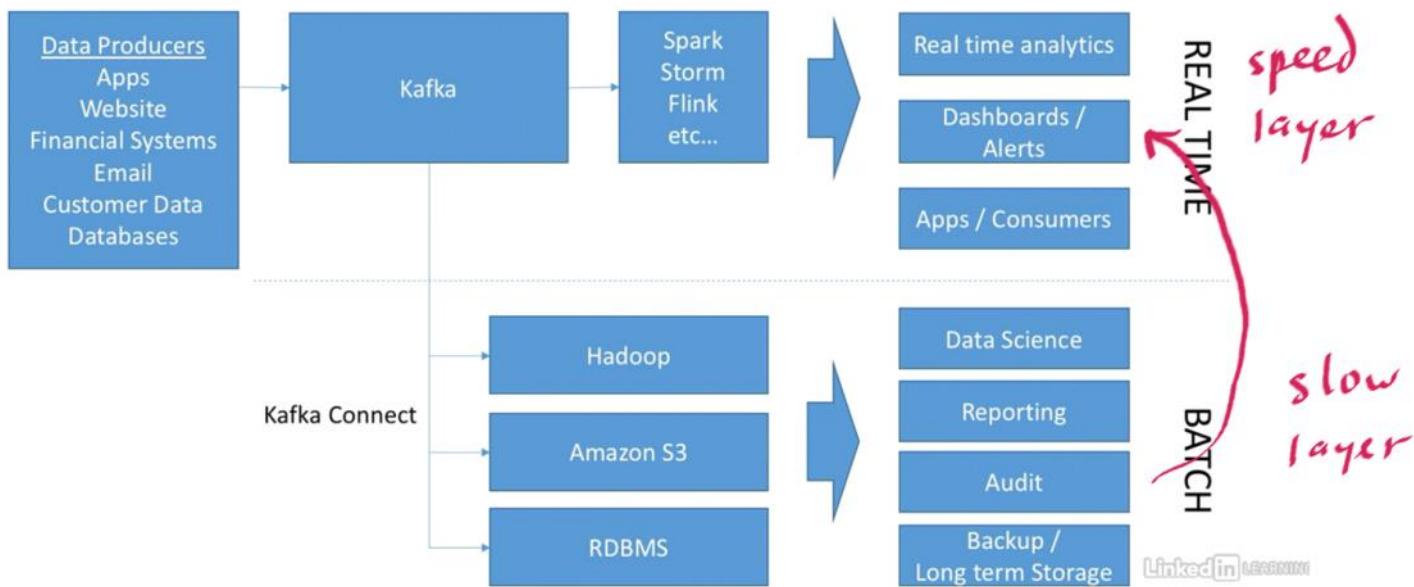
contextual event-driven applications

19 декабря 2020 г. 23:12

real time events + historical/static/reference data

[analytics on event streams](#)

- data ingestion pipeline
- kafka as a front to BigDataIngestion
- паттерн ingestion buffer



отделяют speed layer и slow layer

- It is common to have “generic” connectors or solutions to offload data from Kafka to HDFS, Amazon S3, and ElasticSearch for example
- It is also very common to have Kafka serve a “speed layer” for real time applications, while having a “slow layer” which helps with data ingestions into stores for later analytics

* analytics on event streams

4 февраля 2021 г. 20:53

contextual event-driven applications

early/eager analysis vs late/lazy analysis

- Мы называем это аналитикой при записи, потому что мы выполняем часть анализа нашей работы перед записью в целевое хранилище; вы можете думать об этом как о раннем или нетерпеливом анализе, в то время как аналитика при чтении - это поздний или ленивый анализ.

Table 10.1 Comparing the main attributes of analytics-on-read to analytics-on-write

Analytics-on-read	Analytics-on-write
Predetermined storage format	Predetermined storage format
Flexible queries	Predetermined queries
High latency	Low latency
Support 10–100 users	Support 10,000s of users
Simple (for example, HDFS) or sophisticated (for example, HP Vertica) storage target	Simple storage target (for example, key-value store)
Sophisticated query engine or batch processing framework	Simple (for example, AWS Lambda) or sophisticated (for example, Apache Samza) stream processing framework

можно использовать одновременно оба подхода

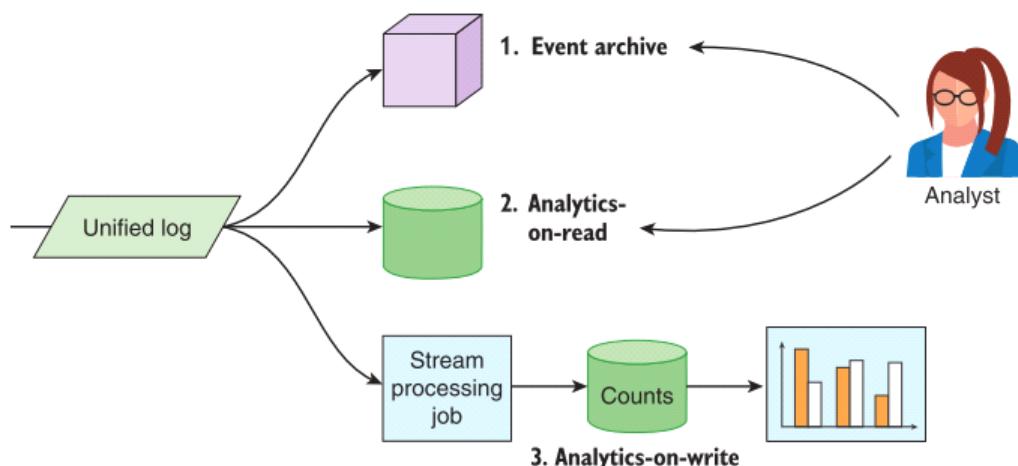


Figure 10.3 Our unified log feeds three discrete systems: our event archive, an analytics-on-read system, and an analytics-on-write system. Event archives were discussed in chapter 7.

гибкость запросов или их скорость

analytics-on-read

- мы сосредоточились на разработке наиболее гибкого способа хранения нашего потока событий в Redshift, чтобы мы могли поддерживать как можно больше разнообразных постфактум- анализов.
- Структура нашего хранилища событий была оптимизирована для обеспечения гибкости в будущем, а не для привязки к какому-либо конкретному анализу, который OOPS может захотеть выполнить позже

analytics-on-write

- мы должны точно понимать , какие аналитики которую OOPS хочет увидеть, чтобы мы могли построить этот анализ для запуска почти в реальном времени в нашем потоке событий Kinesis.

(A) analytics-on-read

- store first; ask questions later
- сначала мы записали наши события в целевое хранилище (S3), и только позже мы выполнили необходимый анализ, когда наше задание Spark прочитало все наши события обратно из нашего архива S3.

Analytics-on-read is really shorthand for a two-step process:

- 1 Write all of our events to some kind of event store.
- 2 Read the events from our event store to perform an analysis.

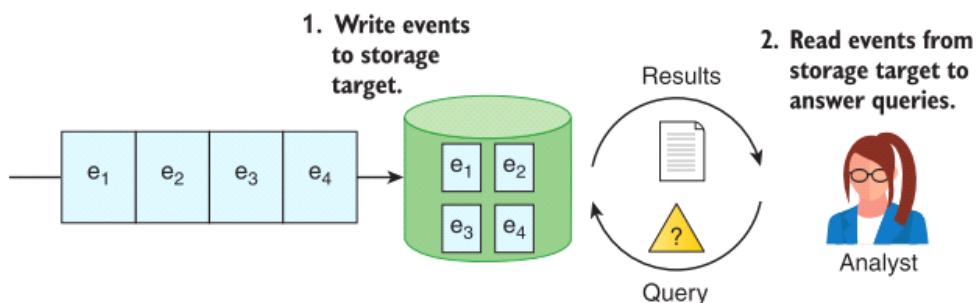


Figure 10.1 For analytics-on-read, we first write events to our storage target in a predetermined format. Then when we have a query about our event stream, we run that query against our storage target and retrieve the results.

analytics-on-read implementation has three key parts:

- A **storage target** to which the events will be written. We use the term storage target because it is more general than the term database.
- A schema, encoding, or format in which the events should be written to the storage target.
- A query engine or data processing framework to allow us to analyze the events as read from our storage target.

пример 1 event warehouse

- у нас есть пять типов событий, каждый из которых отслеживает отдельное действие, включающее подмножество пяти бизнес-сущностей, которые имеют значение для OOPS: грузовики доставки, местоположения, сотрудники, посылки и клиенты.
- Нам необходимо сохранить эти пять типов событий в структуре таблиц в Amazon Redshift с максимальной гибкостью для любой аналитики при чтении, которую мы, возможно, захотим выполнять в будущем

- Delivery truck departs from location at time
- Delivery truck arrives at location at time
- Mechanic changes oil in delivery truck at time

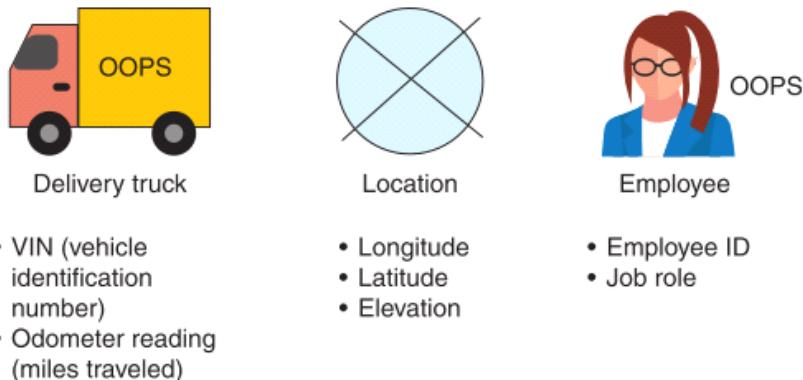
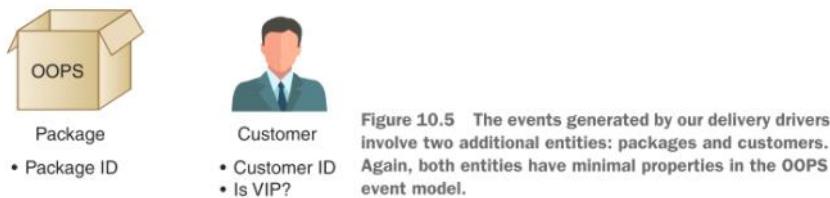


Figure 10.4 The three entities represented in our delivery-truck events are the delivery truck itself, a location, and an employee. All three of these entities have minimal properties—just enough to uniquely identify the entity.

- Driver delivers package to customer at location at time
- Driver cannot find customer for package at location at time



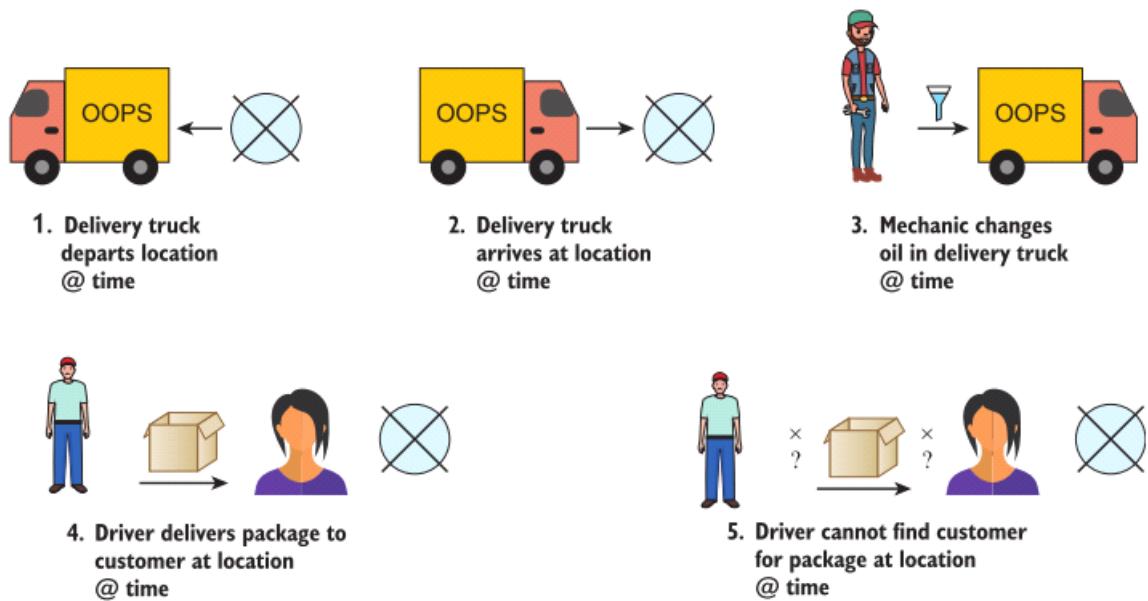


Figure 11.1 The five types of events generated at OOPS are unchanged since their introduction in the previous chapter.

(вариант1) TABLE PER EVENT

- Один наивный подход - создать таблицу для каждого из пяти типов событий
- одну очевидную проблему: для выполнения любого вида анализа по всем типам событий мы должны использовать команду SQL UNION для объединения пяти SELECT вместе
 - o если бы у нас было 30 или 300 типов событий - даже простой подсчет событий в час стал бы чрезвычайно болезненным!
- Такой подход «таблица за событием» имеет вторую проблему: наши пять бизнес-объектов дублируются в нескольких таблицах событий. Например, столбцы для нашего сотрудника дублируются в трех таблицах; столбцы для географического положения находятся в четырех из пяти наших таблиц.
 - o Если, скажем, OOPS решит, что все местоположения также должны иметь почтовые индексы, мы должны обновить четыре таблицы, чтобы добавить новый столбец.

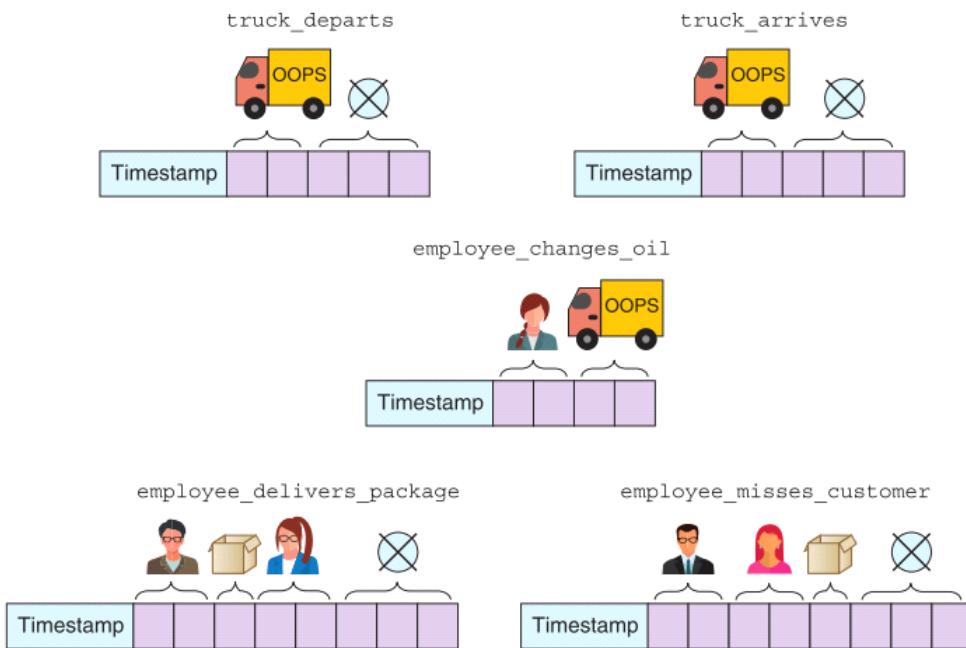


Figure 10.9 Following the table-per-event approach, we would create five tables for OOPS. Notice how the entities recorded in the OOPS events end up duplicated multiple times across the various event types.

(вариант2) FAT TABLE

- Для простой модели событий, такой как OOPS, толстая таблица может работать хорошо: ее легко определить, легко запросить и относительно просто поддерживать.
- Но он начинает показывать свои пределы, когда мы пытаемся моделировать более сложные события;

Что, если мы хотим начать отслеживать события между двумя сотрудниками (например, передачу грузовика)? Нужно ли добавлять в таблицу еще один набор столбцов сотрудников?

Как мы поддерживаем события, включающие набор одной и той же сущности - например, когда сотрудник упаковывает n элементов в пакет?

A *fat table* refers to a single events table, containing just the following:

- A short tag describing the event (or example, TRUCK_DEPARTS or DRIVER_MISSES_CUSTOMER)
- A timestamp at which the event took place
- Columns for each of the entities involved in our events

The table is **sparsely populated**, meaning that columns will be empty if an event does not feature a given entity. Figure 10.10 depicts this approach.

The diagram illustrates a 'Fat events table' where all event types are recorded in a single table. The table structure is as follows:

Event	Timestamp				
Event	Timestamp				
Event	Timestamp				
Event	Timestamp				
Event	Timestamp				
Event	Timestamp				
Event	Timestamp				

Annotations above the table identify the columns:

- Delivery truck**: Points to the first column.
- Location**: Points to the second column.
- Employee**: Points to the third column.
- Package**: Points to the fourth column.
- Customer**: Points to the fifth column.

Vertical dashed lines on the right side of the table indicate the continuation of the table structure from 'Row 1' to 'Row n'.

Figure 10.10 The fat-table approach records all event types in a single table, which has columns for each entity involved in the event. We refer to this table as "sparsely populated" because entity columns will be

Event	Timestamp									Row n
-------	-----------	--	--	--	--	--	--	--	--	-------

Figure 10.10 The fat-table approach records all event types in a single table, which has columns for each entity involved in the event. We refer to this table as “sparsely populated” because entity columns will be empty if a given event did not record that entity.

(вариант3) SHREDDED ENTITIES

- Вместе с этой главной таблицей у нас также есть таблица для каждой сущности; поэтому в случае OOPS у нас было бы пять дополнительных таблиц - по одной для грузовиков доставки, местоположений, сотрудников, пакетов и клиентов.
- Каждая таблица сущностей содержит все свойства для данной сущности, но, что очень важно, она также включает столбец с идентификатором события родительского события, которому эта сущность принадлежит. Эта связь позволяет аналитику ПРИСОЕДИНИТЬ соответствующие сущности к их родительским событиям.

as *shredded entities*. We still have a master table of events, but this time it is “thin,” containing only the following:

- A short tag describing the event (for example, TRUCK_DEPARTS)
- A timestamp at which the event took place
- An event ID that uniquely identifies the event (UUID4s work well)

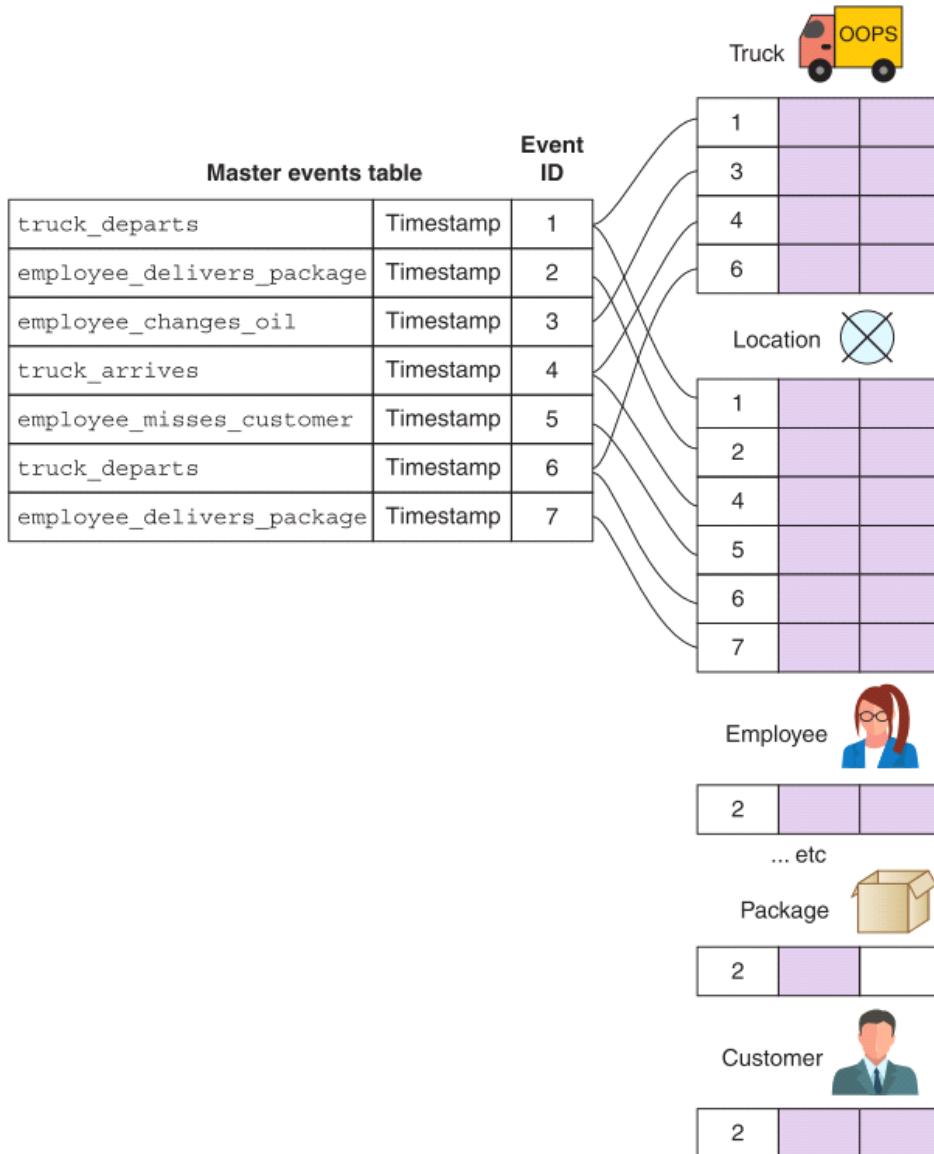


Figure 10.11 In the shredded-entities approach, we have a “thin” master events table that connects via an event ID to dedicated entity-specific tables. The name comes from the idea that the event has been “shredded” into multiple tables.

This approach has advantages:

- We can support events that consist of multiple instances of the same entity, such as two employees or n items in a package.
- Analyzing entities is super simple: all of the data about a single entity type is available in a single table.
- If our definition of an entity changes (for example, we add a zip code to a location), we have to update only the entity-specific table, not the main table.

(Б) analytics- on-write

- Аналитика при записи включает в себя определение нашего анализа заранее и выполнение его в реальном времени по мере поступления событий. Это отлично подходит для информационных панелей, оперативной отчетности и других случаев использования с малой задержкой.

- *Very low latency*—The various dashboards and reports must be fed from the incoming event streams in as close to real time as possible. The reports must not lag more than five minutes behind the present moment.
 - *Supports thousands of simultaneous users*—For example, the parcel tracker on the website will be used by large numbers of OOPS customers at the same time.
 - *Highly available*—Employees and customers alike will be depending on these dashboards and reports, so they need to have excellent uptime in the face of server upgrades, corrupted events, and so on.
 - *Low-latency operational reporting*—This must be fed from the incoming event streams in as close to real time as possible.
 - *Dashboards to support thousands of simultaneous users*—For example, a parcel tracker on the website for OOPS customers.
- is best served by *analytics-on-write*. Analytics-on-write is a four-step process:
- 1 *Read* our events from our event stream.
 - 2 *Analyze* our events by using a stream processing framework.
 - 3 *Write* the summarized output of our analysis to a storage target.
 - 4 *Serve* the summarized output into real-time dashboards or reports.

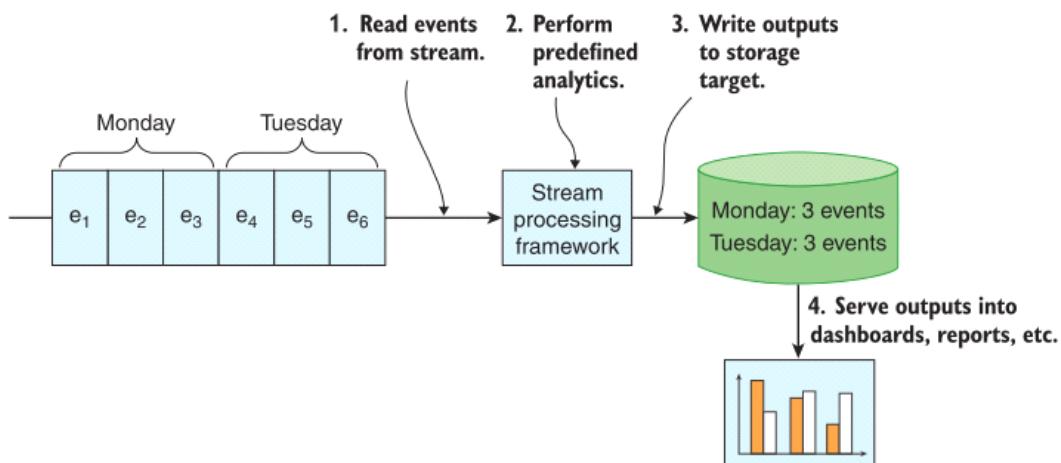


Figure 10.2 With analytics-on-write, the analytics are performed in stream, typically in close to real time, and the outputs of the analytics are written to the storage target. Those outputs can then be served into dashboards and reports.

[аналитика при записи требует предварительных затрат:](#)

- мы должны заранее решить, какой анализ проводить, и разместить этот анализ realtime в нашем потоке событий.
- Взамен этого ограничения мы получаем некоторые преимущества: наши запросы имеют низкую задержку, могут обслуживать множество одновременных пользователей и просты в эксплуатации.
- Чтобы реализовать аналитику при записи для OOPS, нам понадобится база данных для хранения наших агрегатов и фреймворк потоковой обработки для преобразования наших событий в агрегаты

[пример 2](#)

- нам нужен доступ к потоку событий OOPS, когда он проходит через Amazon Kinesis почти в реальном времени.
- точно выяснить, какую аналитику в режиме реального времени они хотят видеть.
 - The location of each delivery truck
 - The number of miles each delivery truck has driven since its last oil change

- 1** Read each event from Kinesis.
- 2** If the event involves a delivery truck, use its current mileage to update the count of miles since the last oil change.
- 3** If the event is an oil change, reset the count of miles since the last oil change.
- 4** If the event associates a delivery truck with a specific location, update the table row for the given truck with the event's latitude and longitude.

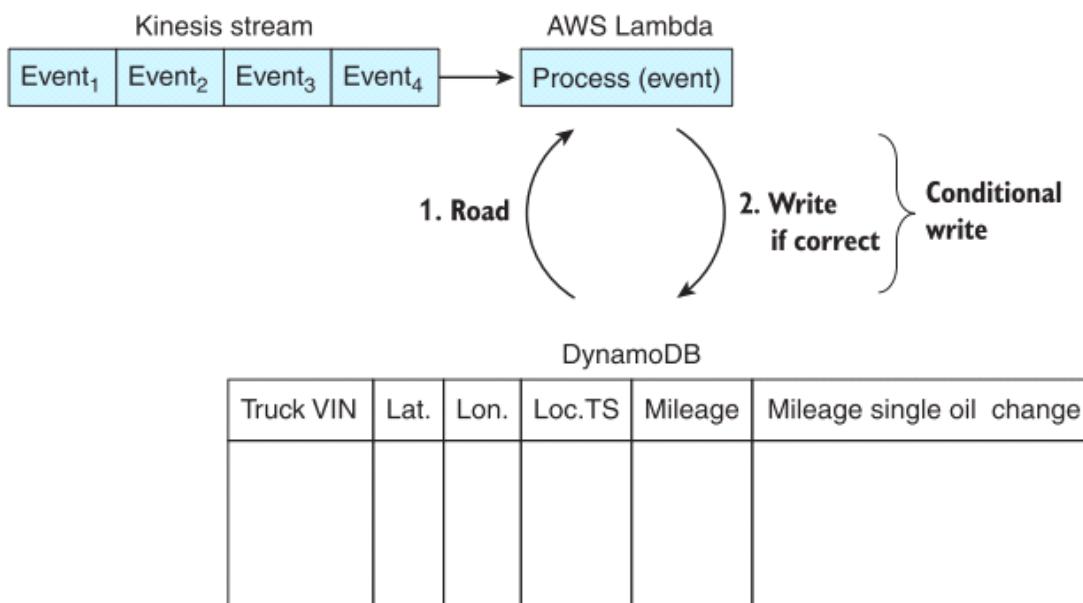


Figure 11.6 Our AWS Lambda function will read individual events from OOPS, check whether our table in DynamoDB can be updated with the event's values, and update the row in DynamoDB if so. This approach uses a DynamoDB feature called conditional writes.

сначала построим простую модель событий

Table 11.1 The status of OOPS's delivery trucks

Truck VIN	Latitude	Longitude	Miles since oil change
1HGCM82633A004352	51.5208046	-0.1592323	35
JH4TB2H26CC000000	51.4972997	-0.0955459	167
19UYA31581L000000	51.4704679	-0.1176902	78

добавим ключ упорядочивания событий (если мы НЕ используем упорядочивание в партиции в кафке)

- Теперь, когда мы обрабатываем событие от Kinesis, содержащее местоположение грузовика, мы сравниваем временную метку этого события с существующей меткой времени местоположения в нашей таблице. Только если метка времени нового события «превосходит» существующую метку времени (она более поздняя), мы обновляем широту и

долготу нашего грузовика.

Table 11.2 Adding the *Location timestamp* column to our table

Truck VIN	Latitude	Longitude	Location timestamp	Miles since oil change
1HGCM8...	51.5208046	-0.1592323	2018-08-02T21:50:49Z	35
JH4TB2...	51.4972997	-0.0955459	2018-08-01T22:46:12Z	167
19UYA3...	51.4704679	-0.1176902	2018-08-02T18:14:45Z	78

как рассчитать *stateful* показатель

- Уловка состоит в том, чтобы понять, что мы не должны хранить эту метрику в нашей таблице.
- Вместо этого мы должны сохранить два входных значения, которые используются для расчета этой метрики
- Помните, что для того, чтобы эта таблица была точной, мы всегда хотим записывать последний пробег грузовика и пробег грузовика при последней известной замене масла.
 - Current (latest) mileage
 - Mileage at the time of the last oil change

`miles since oil change = current mileage - mileage at last oil change`

Table 11.3 Replacing our *Miles since oil change* metric with *Mileage* and *Mileage at oil change*

Truck VIN	Latitude	Longitude	Location timestamp	Mileage	Mileage at oil change
1HGCM8...	51.5...	-0.15...	2018-08-02T21:50:49Z	12453	12418
JH4TB2...	51.4...	-0.09...	2018-08-01T22:46:12Z	19090	18923
19UYA3...	51.4...	-0.11...	2018-08-02T18:14:45Z	8407	8329

добавим упорядочивание для пробега

- Помните, что для того, чтобы эта таблица была точной, мы всегда хотим записывать последний пробег грузовика и пробег грузовика при последней известной замене масла.
- Опять же, как нам справиться с ситуацией, когда более раннее событие прибывает после более недавнего события?
- На этот раз мы имеем кое-что в нашу пользу, а именно то, что пробег грузовика со временем монотонно увеличивается
- более высокий пробег всегда является более новым, и мы можем использовать это правило, чтобы однозначно отбросить пробег более старых событий.

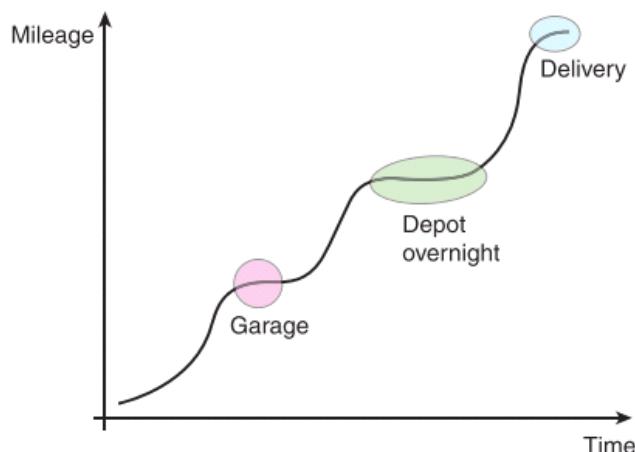


Figure 11.4 The mileage recorded on a given truck's odometer monotonically increases over time. We can see periods where the mileage is flat, but it never decreases as time progresses.

как сделать лямбда-функцию

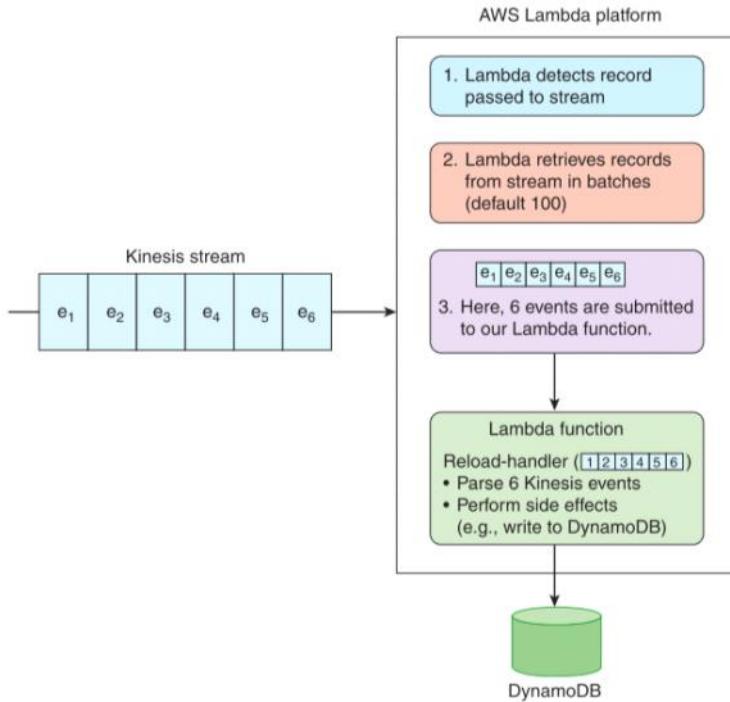


Figure 11.7 AWS Lambda detects records posted to a Kinesis stream, collects a microbatch of those records, and then submits that microbatch to the specified Lambda function.

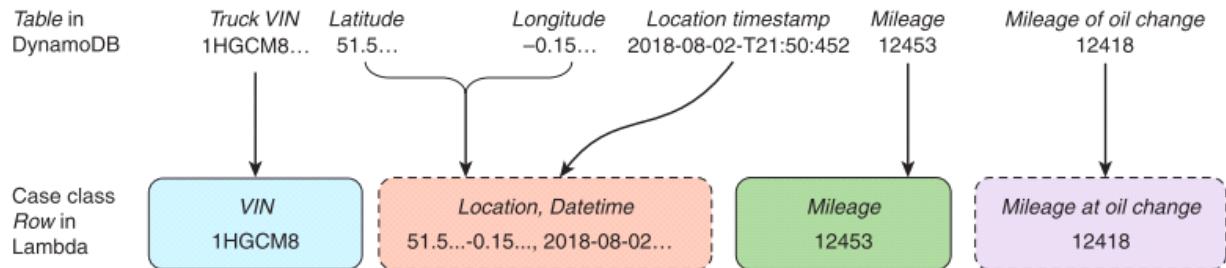


Figure 11.10 The Row case class in our Lambda function will contain the same data points as found in our DynamoDB table. The dotted lines indicate that a given Row instance may not contain these data points, because the related event types were not found in this microbatch for this OOPS truck.

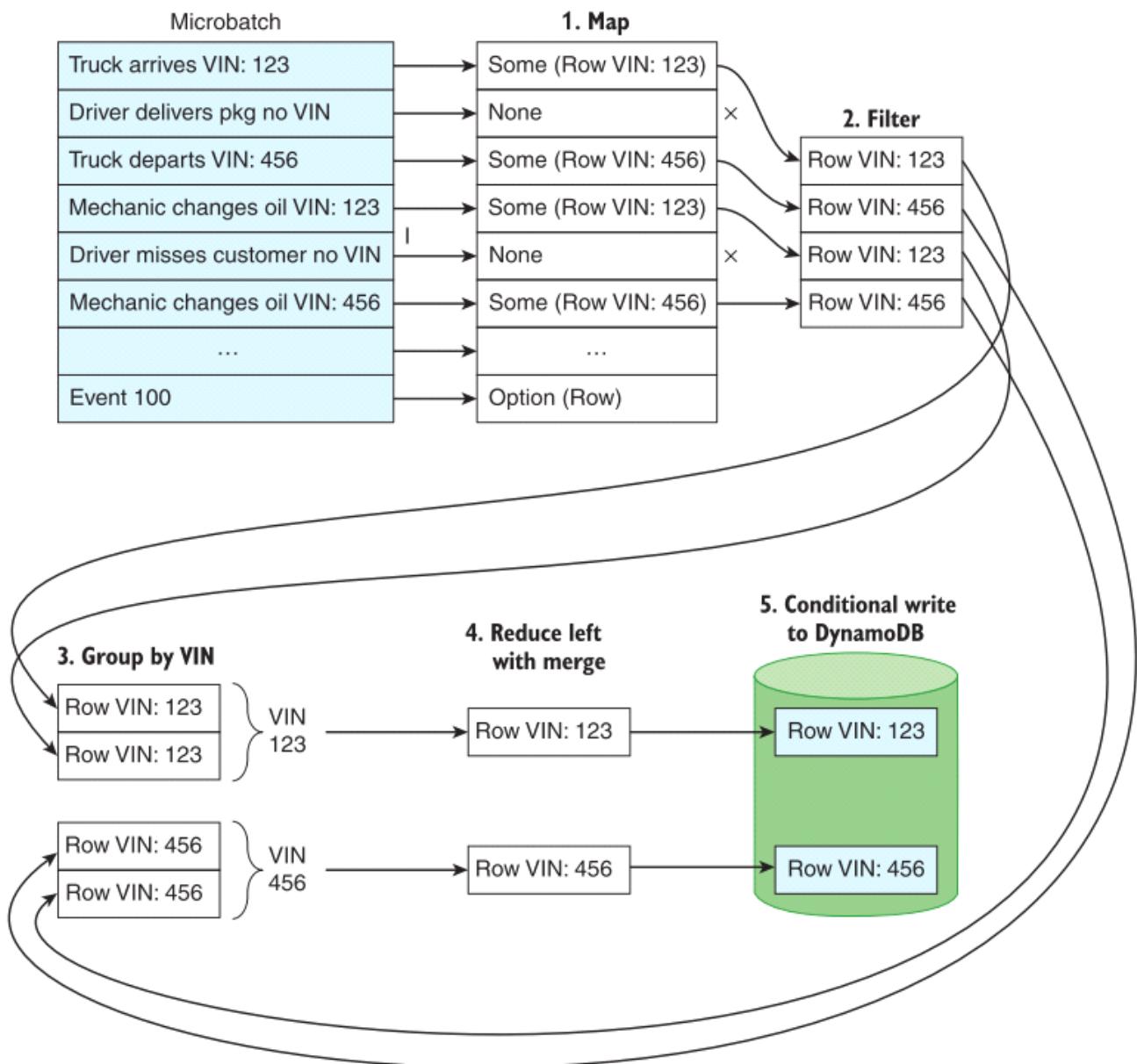


Figure 11.11 First, we map the events from our microbatch into possible Rows, and filter out the Nones. We then group our Rows by the truck's VIN and reduce each group to a single Row per truck by using a merge. Finally, we perform a conditional write for each Row against our table in DynamoDB.

(вариант1) conditional write for every single event [in batch](#)

- обработки одного события за раз,
- Конечно, мы могли бы применить наш алгоритм к каждому событию в микропакете
- К сожалению, этот подход является расточительным, если одна микропакет содержит

несколько событий, относящихся к одному и тому же грузовику OOPS, что может легко произойти . мы хотим обновить нашу таблицу DynamoDB с учетом последнего (самого высокого) пробега данного грузовика? Если наш микропакет содержит 10 событий, относящихся к Truck 123, наивная реализация нашего алгоритма из рисунка 11.8 потребует 10 условных записей в таблицу DynamoDB только для обновления строки Truck 123. Чтение и запись в удаленные базы данных - дорогостоящая операция, поэтому мы должны стремиться минимизировать ее в нашей лямбда-функции

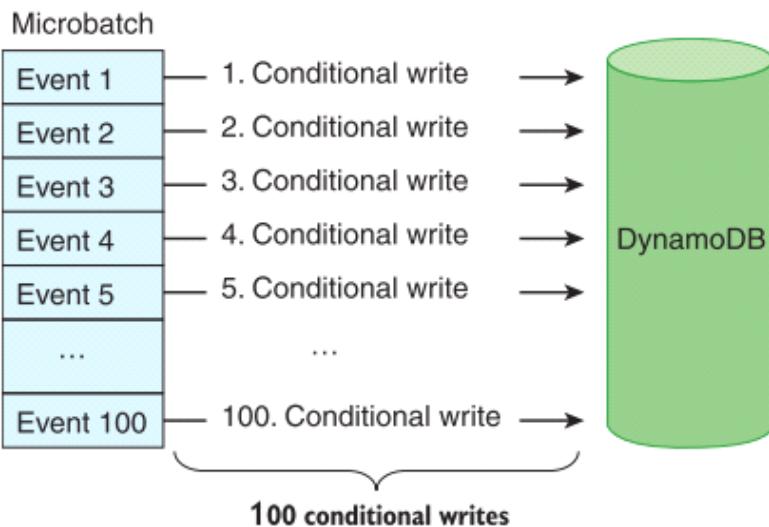


Figure 11.8 A naïve approach to updating our DynamoDB table from our microbatch of events would involve a **conditional write** for every single event.

(вариант 2) applying a pre-aggregation to our microbatch of events

- Решение состоит в том, чтобы предварительно агрегировать наш микропакет событий внутри нашей лямбда-функции.
- Имея 10 событий в микропакете, относящемся к Truck 123, нашим первым шагом должно быть определение максимального пробега по этим 10 событиям. Этот наивысший показатель пробега - единственный, который нам нужно попытаться записать в DynamoDB; Нет никакого смысла беспокоить DynamoDB девятью меньшими милями.
- При этом **предварительная агрегация - это относительно дешевый процесс в памяти, поэтому всегда стоит пытаться это сделать**, даже если мы предотвратим лишь несколько ненужных операций DynamoDB

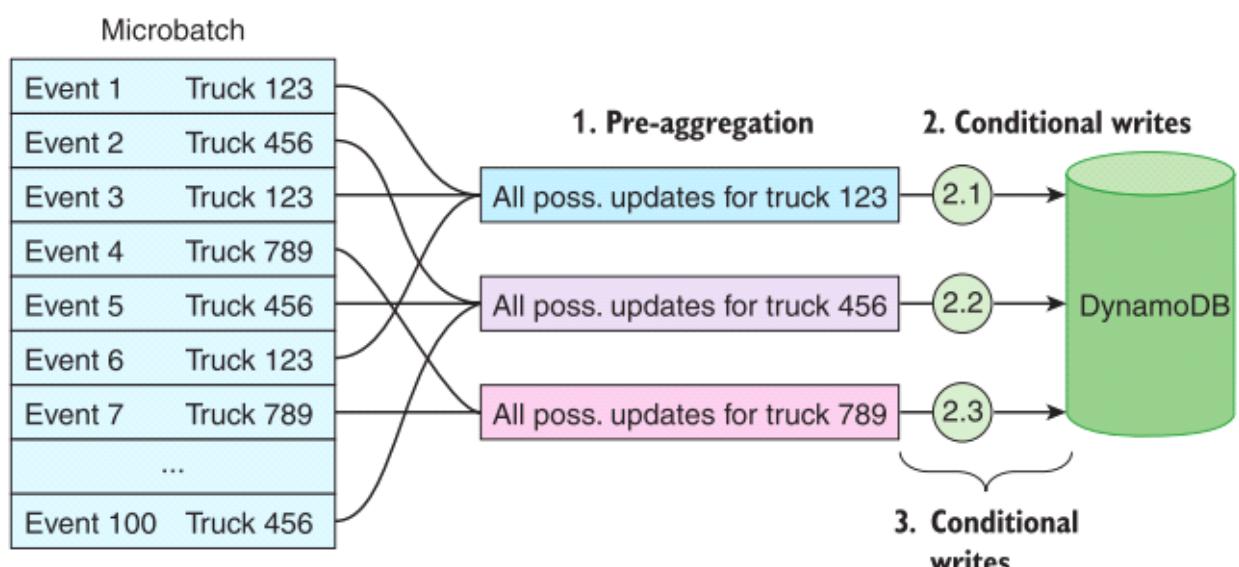


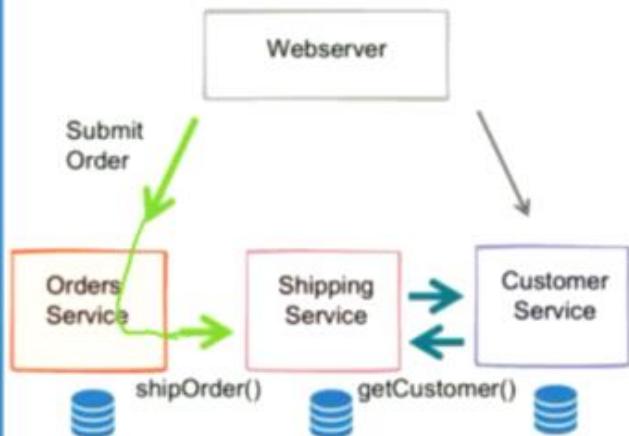
Figure 11.9 By applying a pre-aggregation to our microbatch of events, we can reduce the required number of conditional writes down to one per OOPS truck found in our microbatch.

пример 1 event carried state transfer

21 декабря 2020 г. 20:16

Buying an iPad (with REST)

- Orders Service calls Shipping Service to tell it to ship item.
- Shipping service looks up address to ship to (from Customer Service)



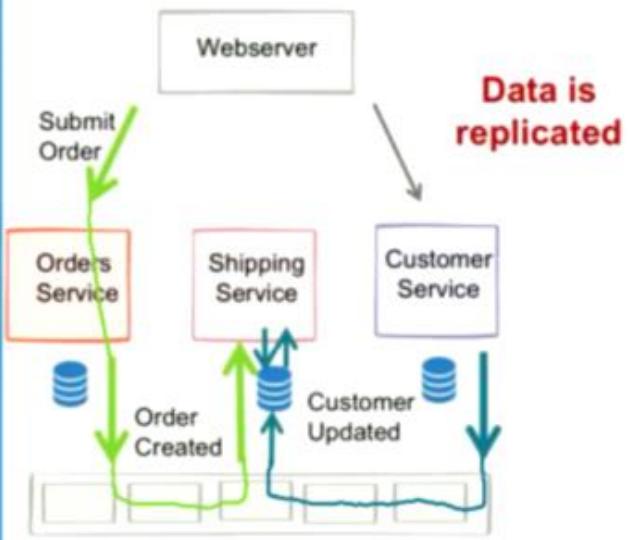
Using events for Notification

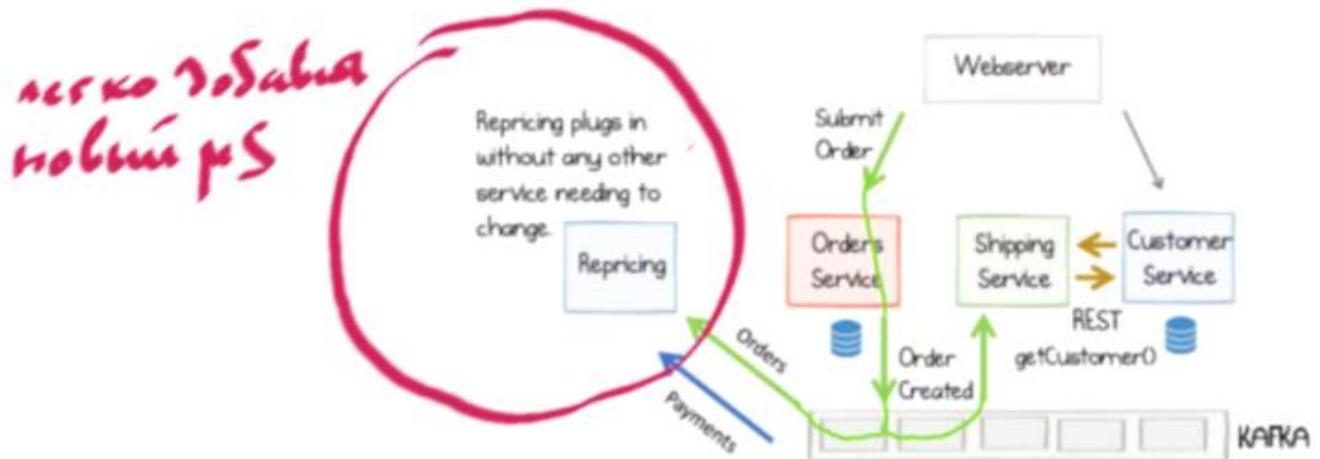
- Orders Service no longer knows about the Shipping service (or any other service). Events are fire and forget.



Using events to share facts

- Call to Customer service is gone.
- Instead data is replicated, as events, into the shipping service, where it is queried locally..





пример 2 event carried state transfer

21 декабря 2020 г. 20:17

В итоге сервис заказа, всегда держит все состояние запасов товаров

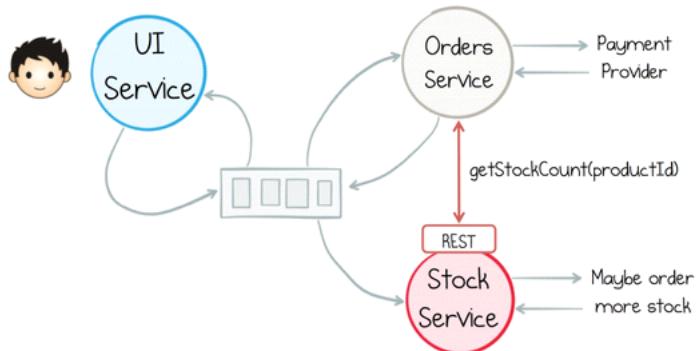
- ТК микросервис всегда держит актуальное состояние то он по-сущи stateful
- Сервисы по своей сути становятся отслеживающими состояние. Им необходимо отслеживать и курировать распространяемый набор данных с течением времени. Дублирование состояния может также усложнить рассуждение о некоторых проблемах (как атомарно уменьшить количество запасов?)

<https://www.confluent.io/blog/build-services-backbone-events/>

Blending Events and Queries

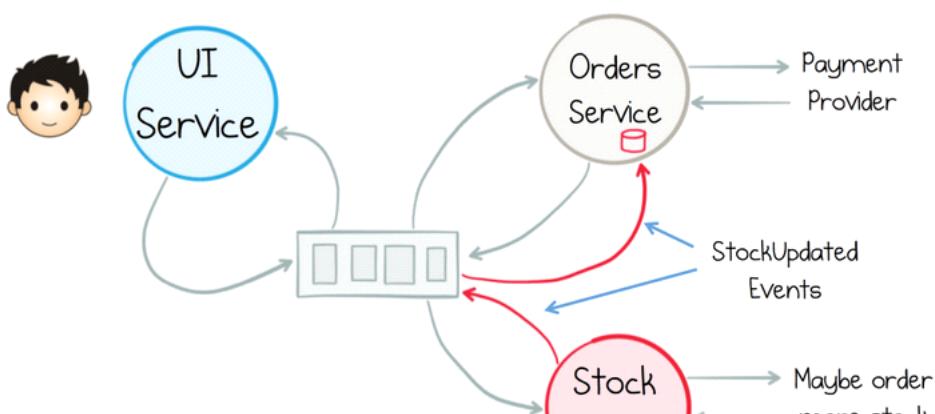
The above example considered only commands/events. There were no queries (remember we defined all interactions as being one of commands, events and queries earlier). Queries are a necessity for all but the simplest architecture. So let's extend the example a bit, making the Orders Service check that there is sufficient Stock before it processes a Payment.

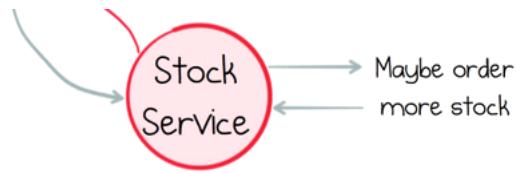
The request-driven approach to this would involve sending a query to the Stock service to retrieve the current stock count. This leads to a hybrid model, where the event stream is used purely for notification, allowing any service to dip into the flow, but queries go directly to source.



For larger ecosystems, where services need to evolve independently, remote queries add a lot of coupling, tying services together at runtime. We can avoid such cross-context queries by internalising them. The stream of events is used to cache datasets in each service, where they can be queried locally.

So to add this stock check, the Orders Service would subscribe to the stream of Stock events, storing them locally in a database. It would then query this 'view' to validate there is sufficient stock.





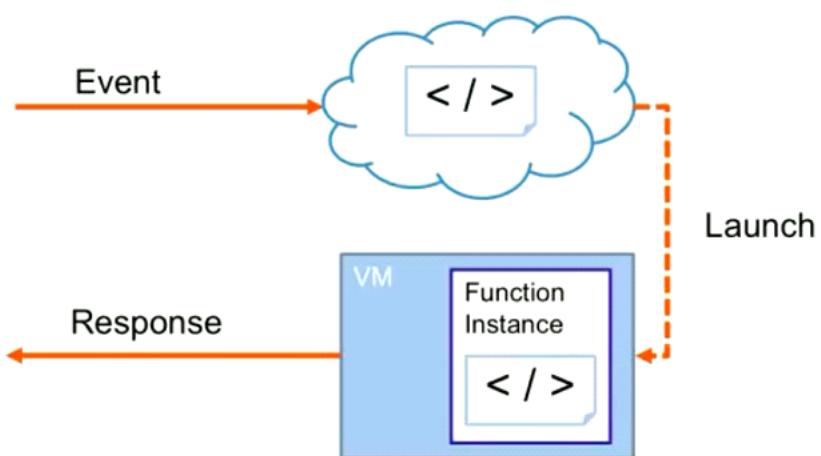
Pure event-driven systems have no concept of remote queries – events propagate state to services where it is queried locally

There are three advantages of this 'Query by Event-Carried State Transfer' approach:

1. Better decoupling: Queries are local. They involve no cross-context calls. This ties services far less tightly than their request-driven brethren.
2. Better autonomy: The Orders service has a private copy of the Stock dataset so it can do whatever it likes with it, rather than being limited to the query functionality offered by the Stock Service.
3. Efficient Joins: If we were to "look up the stock" on every order, we would effectively be doing a join over the network between the two services. As workloads grow, or more sources need to be combined, this can be increasingly arduous. 'Query by Event Carried State Transfer' solves this issue by bringing queries (and joins) local.

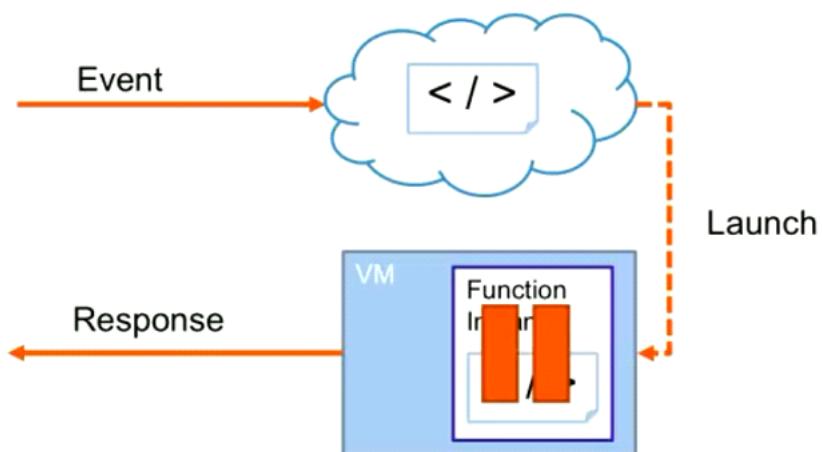
This approach isn't without its downsides though. Services become inherently stateful. They need to keep track of, and curate, the propagated data set over time. The duplication of state can also make some problems harder to reason about (how do we decrement the stock count atomically?) and we should be careful of divergence over time. But all these issues have workable solutions, they just need a little consideration. This is well worth the effort for larger, more complex estates.

Function as a Service

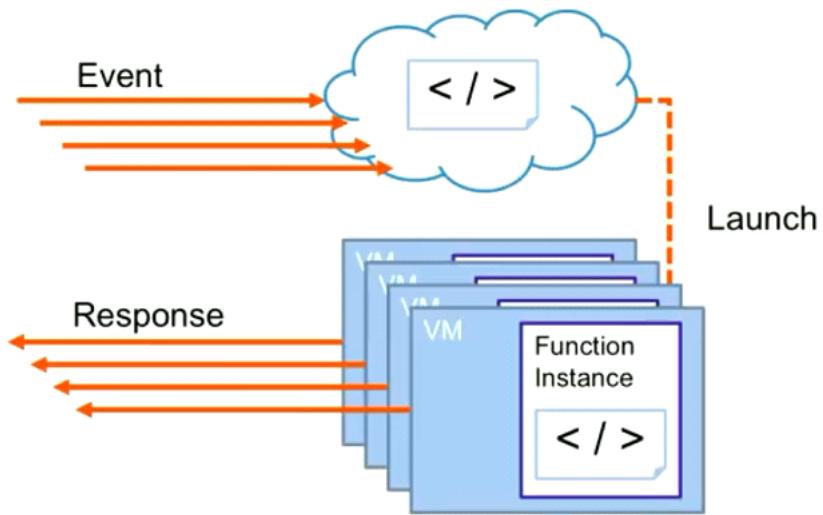


Смысль в том чтобы платить меньше, если событий нет

When nothing happens

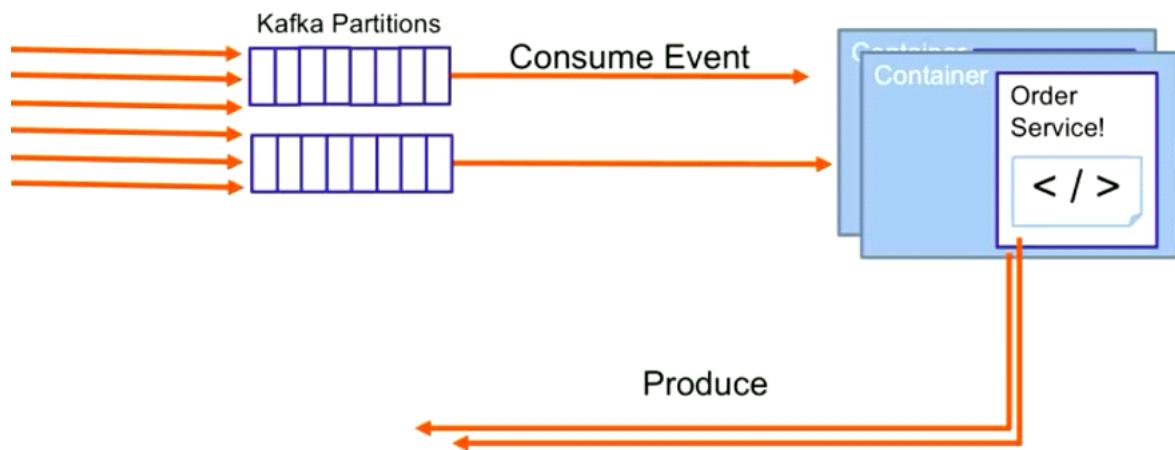


At scale



где место кафки в этом

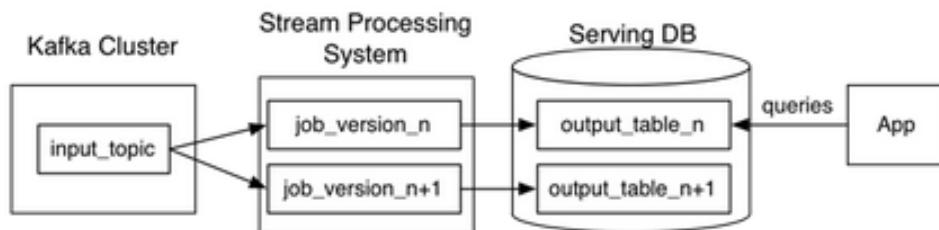
Wait, this is super familiar



streaming architecture

- потоковая обработка кажется неуместной для высокопроизводительной обработки исторических данных.
- Используйте Kafka или какую-либо другую систему, которая позволит вам сохранить полный журнал данных, которые вы хотите обрабатывать повторно
- если вы хотите повторно обрабатывать данные за 30 дней, установите срок хранения в Kafka на 30 дней

1. Use Kafka or some other system that will let you retain the full log of the data you want to be able to reprocess and that allows for multiple subscribers. For example, if you want to reprocess up to 30 days of data, set your retention in Kafka to 30 days.
2. When you want to do the reprocessing, start a second instance of your stream processing job that starts processing from the beginning of the retained data, but direct this output data to a new output table.
3. When the second job has caught up, switch the application to read from the new table.
4. Stop the old version of the job, and delete the old output table.

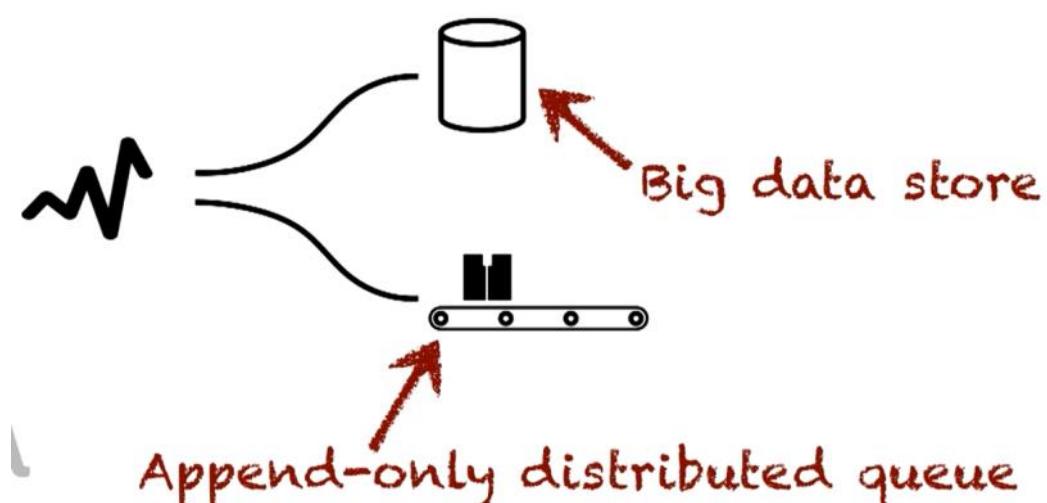
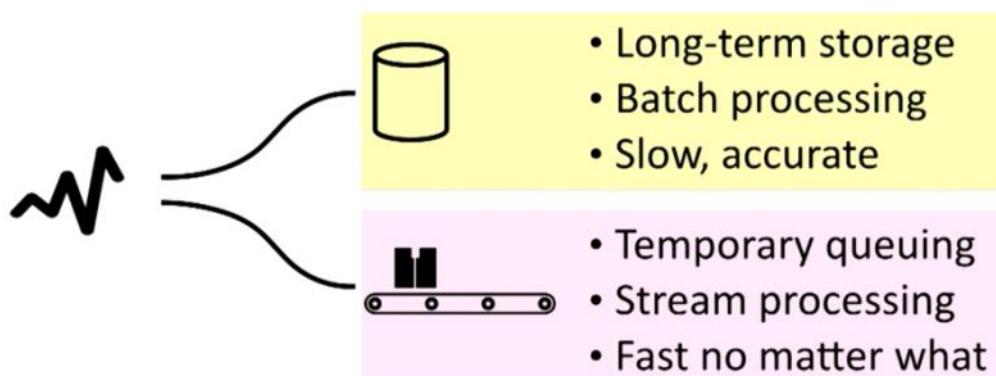


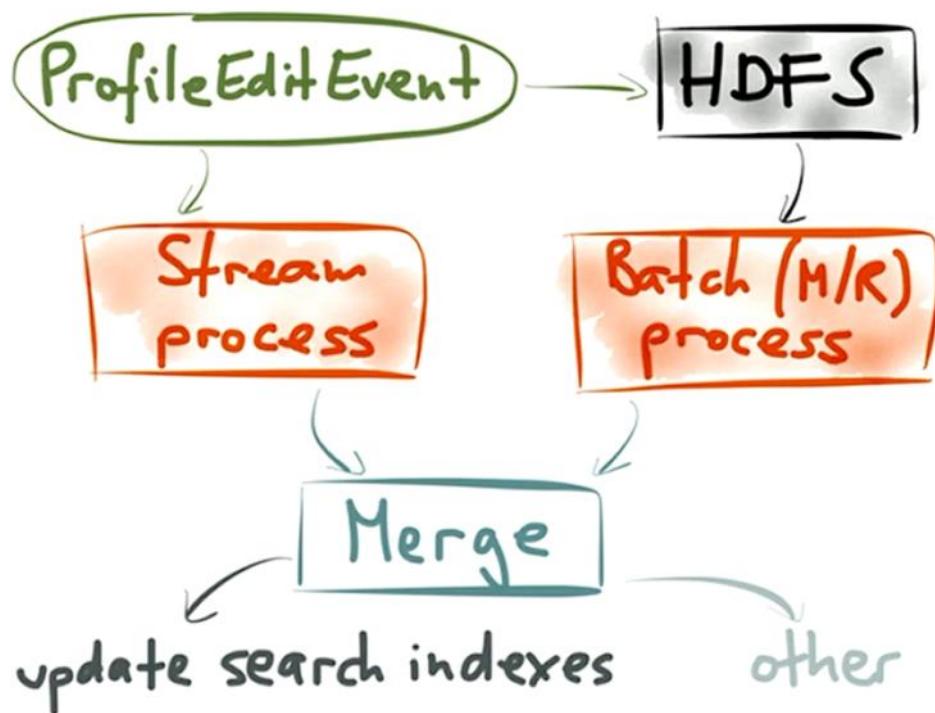
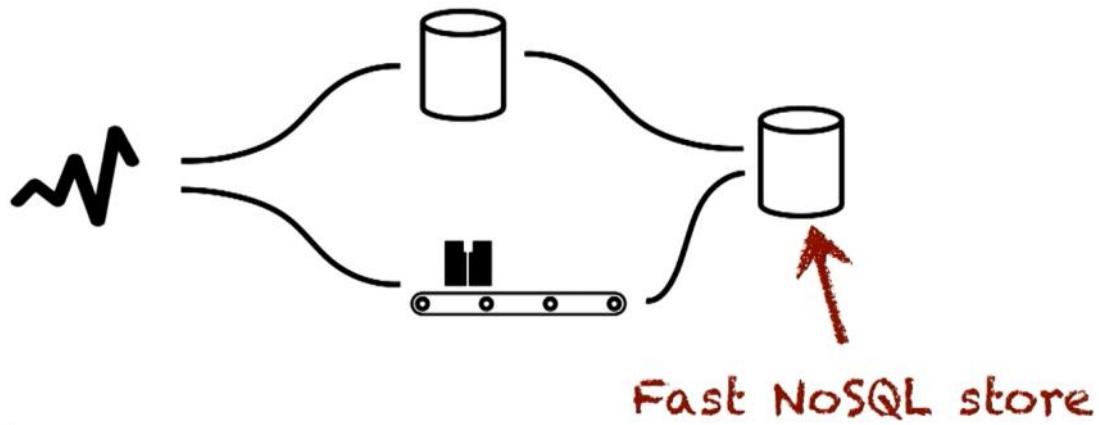
плюсы

- В отличие от лямбда-архитектуры, в этом подходе вы выполняете повторную обработку только при изменении кода обработки, и вам действительно нужно пересчитать результаты.

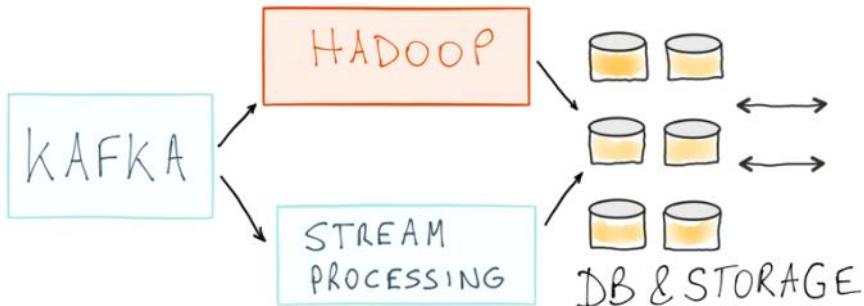
Lambda Architecture

- System input is an event stream
- Events are immutable!
- Batch and stream processing are functional





Lambda архитектура



If you want stream processing and continuous position updates, you would then add a stream processing layer as shown in the lower part of the diagram. You would still typically keep the Hadoop cluster to periodically check and validate the results from the streaming layer or to do data reprocessing if the logic of your computation changes. This hybrid architecture is known as the Lambda architecture. We have already written about the inefficiencies that stem from having two separate clusters that need to be deployed, monitored and debugged. So in summary, there are lots of moving parts and inefficiencies in how things are done today, including:

- An extra Hadoop cluster to reprocess data.
- Storage maintained at the stream processing layer. This is needed for lookups and aggregations (e.g., for keeping track of the number of times an asset is bought or sold).
- Storage and databases maintained for outputs from the streaming and Hadoop jobs.
- Write amplification is inevitable. A record is written internally in the stream processing layer to maintain computational state. This state is also duplicated externally so that it can be queried by other apps.
- Locality is destroyed, because data that needs to be local to processing is unnecessarily shipped to a remote storage cluster.

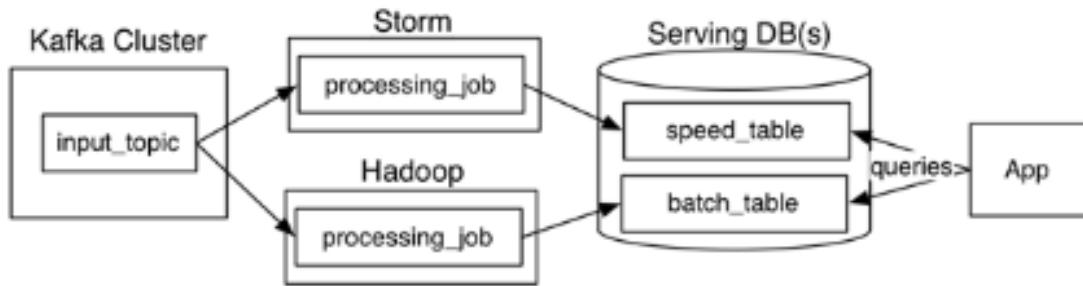
- основная идея состоит в том, что вы запускаете потоковую систему вместе с пакетной системой, выполняя по существу одинаковые вычисления.
 - Система потоковой передачи дает вам неточные результаты с малой задержкой (либо из-за использования алгоритма аппроксимации, либо из-за того, что сама потоковая система не обеспечивает правильности),
 - а через некоторое время система пакетной обработки запускается и предоставляет вам правильный результат.
- Первоначально предложено Натаном Марцем из Twitter (создателем Storm), она оказалась весьма успешной, потому что это была, по сути, фантастическая идея для того времени; потоковые движки были немного разочарованы в отделе корректности, а пакетные движки были по своей сути громоздкими, как и следовало ожидать, поэтому Lambda дала вам возможность съесть свой пресловутый торт и съесть его тоже.
- К сожалению, поддержка системы Lambda является проблемой: вам нужно создать, подготовить и поддерживать две независимые версии вашего конвейера, а затем также каким-то образом объединить результаты из двух конвейеров в конце.

Лямбда архитектура появилась из за того что streaming platform раньше давала неточные результаты

- Само собой разумеется, что для многих пользователей любой риск потери записей или потери данных в конвейерах обработки данных неприемлем. Тем не менее, исторически многие универсальные потоковые системы не давали никаких гарантий относительно обработки записей - вся обработка выполнялась только «по максимуму». Другие системы предоставляли гарантии как минимум один раз, гарантировая, что записи всегда обрабатывались хотя бы один раз, но записи могли дублироваться (и, таким образом, приводить к неточному агрегированию); на практике многие такие системы, которые выполнялись по крайней мере один раз, выполняли агрегирование в памяти, и, таким образом, их агрегаты все еще могли быть потеряны при сбое машин. Эти системы использовались для получения спекулятивных результатов с малой задержкой, но, как правило, ничего не могли гарантировать относительно достоверности этих результатов.
- Как указано в главе 1, это привело к стратегии, которая была придумана Lambda Architecture - запускать потоковую систему для получения быстрых, но неточных результатов.

immutable sequence of records is captured and fed into a batch system and a stream processing system in parallel.

- Лямбда-архитектура предназначена для приложений, построенных на основе сложных асинхронных преобразований, которые должны выполняться с небольшой задержкой (скажем, от нескольких секунд до нескольких часов).



Плюсы

- Lambda Architecture делается упор на сохранение входных данных без изменений.
- Мне также нравится, что эта архитектура подчеркивает проблему повторной обработки данных. Повторная обработка - одна из ключевых проблем потоковой обработки, но ее очень часто игнорируют. Под «повторной обработкой» я имею в виду повторную обработку входных данных для получения выходных данных
- В их распоряжении две вещи, которые не совсем решают их проблему: масштабируемая пакетная система с высокой задержкой, которая может обрабатывать исторические данные, и система обработки потоков с низкой задержкой, которая не может повторно обрабатывать результаты. Склейв эти две вещи вместе, они действительно могут создать рабочее решение.

минусы

- The problem with the Lambda Architecture is that maintaining code that needs to produce the same result in two complex distributed systems is exactly as painful as it seems like it would be. I don't think this problem is fixable.
- В наши дни я советую использовать фреймворк пакетной обработки, такой как MapReduce, если вы не чувствительны к задержкам, и фреймворк потоковой обработки, если он есть, но не пытайтесь делать и то, и другое одновременно, если в этом нет крайней необходимости.
people who tried this experienced a number of issues with the Lambda Architecture:

Inaccuracy

Users tend to underestimate the impact of failures. They often assume that a small percentage of records will be lost or duplicated (often based on experiments they ran), and are shocked on that one bad day when 10% (or more!) of records are lost or are duplicated. In a sense, such systems provide only “half” a guarantee—and without a full one, anything is possible.

Inconsistency

The batch system used for the end-of-day calculation often has different data semantics than the streaming system. Getting the two pipelines to produce comparable results proved more difficult than initially thought.

Complexity

By definition, Lambda requires you to write and maintain two different codebases. You also must run and maintain two complex distributed systems, each with different failure modes. For anything but the simplest of pipelines, this quickly becomes overwhelming.

Unpredictability

In many use cases, end users will see streaming results that differ from the daily results by an uncertain amount, which can change randomly. In these cases, users will stop trusting the streaming data and wait for daily batch results instead, thus destroying the value of getting low-latency results in the first place.

Latency

Some business use cases require low-latency correct results, which the Lambda Architecture does not provide by design.

главное преимущество лямбда-архитектуры в ее скорости

- Lambda Architecture стала довольно популярной, несмотря на связанные с ней затраты и

головную боль, просто потому, что она удовлетворила критическую потребность, которую многие компании в противном случае испытывали затруднительно: получение низкой задержки, но в конечном итоге исправить результаты из своих конвейеров обработки данных.

* kappa архитектура (на основе КАФКИ!!)

7 февраля 2021 г. 22:24

kappa архитектура

- означает запуск одного конвейера с использованием хорошо спроектированной системы, которая должным образом построена для работы под рукой.
- В качестве промежуточного шага давайте упростим описанное выше развертывание, удалив уровень Hadoop и выполнив всю обработку на уровне потоковой передачи. Итак, мы переходим от архитектуры Lambda к так называемой архитектуре Карпа

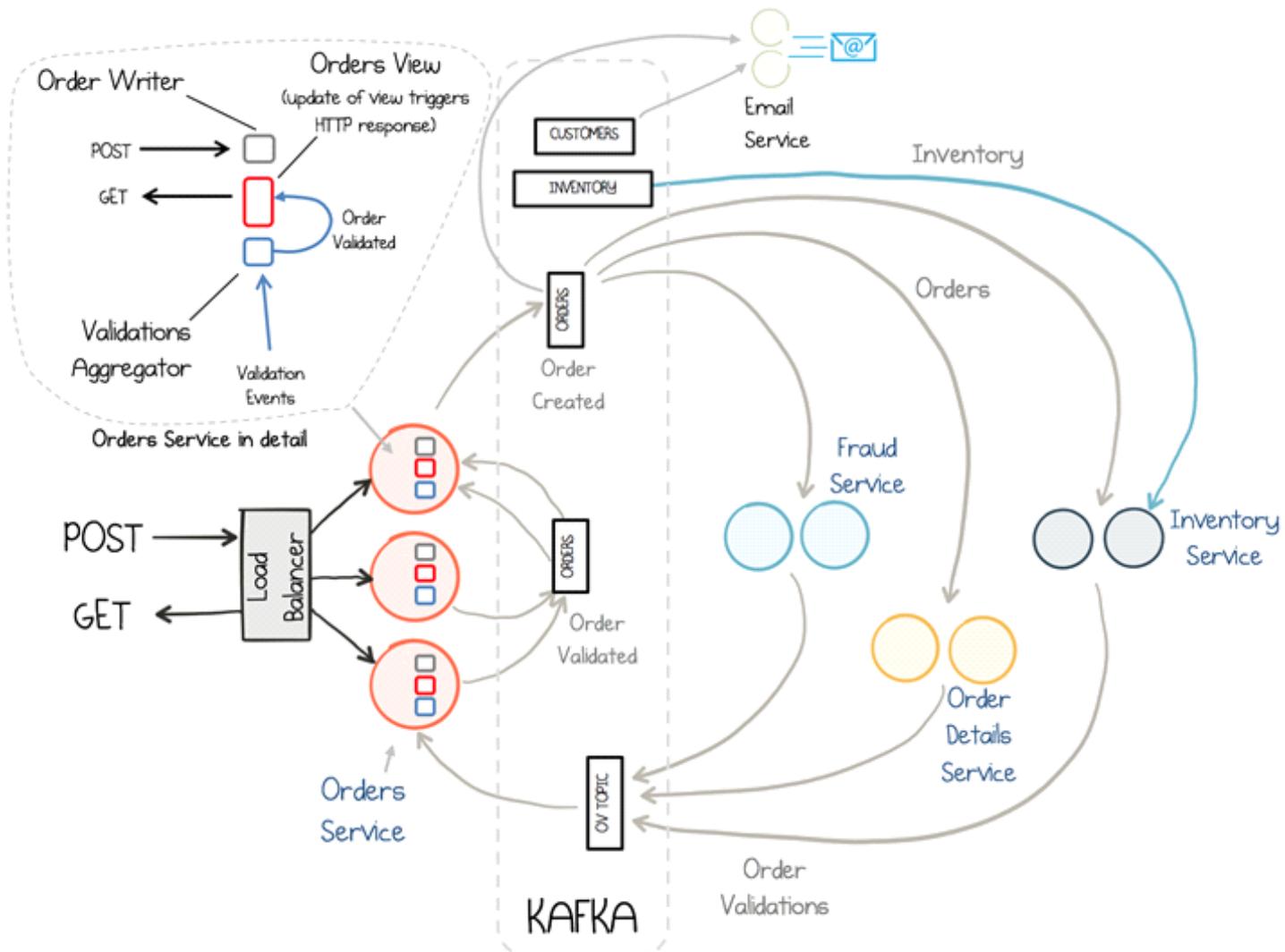
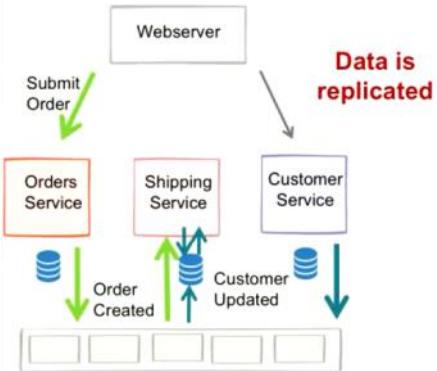
K architecture
"process real-time data and
re-process historical data
in the same framework"



пример SMALL (QCon 2018 Gwen Shapira)

20 декабря 2020 г. 18:53

<https://github.com/confluentinc/qcon-microservices>
<https://drive.google.com/file/d/1vg1tclk4FWK7BKRG5cEhOzbeXov65hUK/view>
[Beyond Microservices: Streams, State and Scalability](#)



... сервис 1

20 декабря 2020 г. 21:24

(2) сервис размещения заказов от юзера (выбрали Option #2 см картинку ниже)

также используется long polling (см Collapsing CQRS with a Blocking Read паттерн)

- Perform a "Long-Poll" styled get. This method will attempt to get the value for the passed key blocking until the key is available or passed timeout is reached. Non-blocking IO is used to implement this, but the API will block the calling thread if no data is available
- [onenote:///C:/Users/trans/Qsync/vova_from_onenote/tf_agonote_v1/SYSTEMDESIGN/KAFKA_LOCAL%20STATE%20STORE.one#%20blocking%20read%20pattern\(c%20помощью%20long%20polling\)§ion-id=%720F6F5D-93F8-4266-949D-E246C971A22&page-id=113D1C6D0-94D5-4D77-9EBE-377721365BB5&end](http://C:/Users/trans/Qsync/vova_from_onenote/tf_agonote_v1/SYSTEMDESIGN/KAFKA_LOCAL%20STATE%20STORE.one#%20blocking%20read%20pattern(c%20помощью%20long%20polling)§ion-id=%720F6F5D-93F8-4266-949D-E246C971A22&page-id=113D1C6D0-94D5-4D77-9EBE-377721365BB5&end)

используется встроенный в kafka streams: локальный state store RocksDB

- те стримсы использую не для трансформаций, а просто для того чтобы читать из топика и класть в локальную БД (а для того чтобы читать из этой БД у kafka streams есть отдельный key-value API)
- RocksDB умеет "из-коробки" автоматически себя восстанавливать, в том случае, если вдруг пропадет
- я могу явно задать папку на диске для хранения RocksDB, например persistent volume в кубере

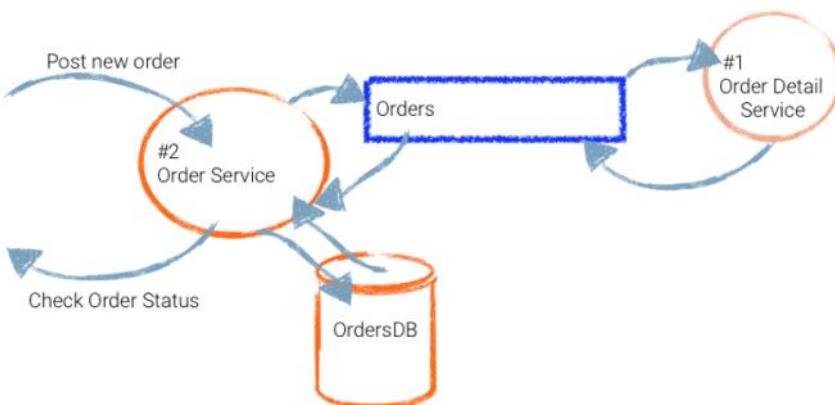
основной тред проги создает веб-сервер

- а дополнительный тред

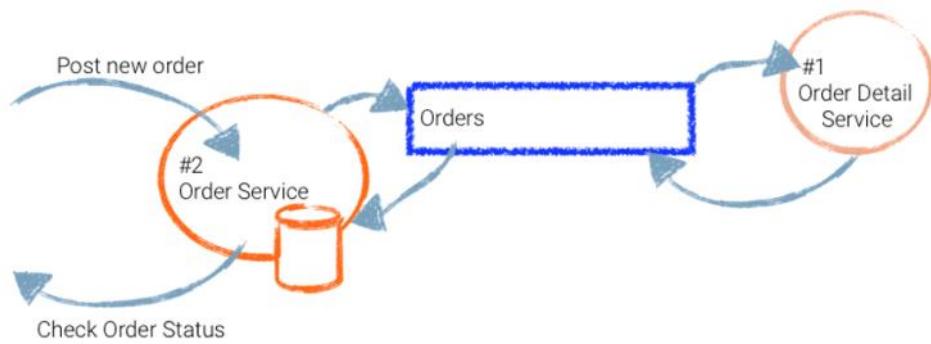
возможно потребуется поиск другого стримс-клиента если данного ключа в данной партиции не оказалось

- тогда через кафка-метаданные ищем хост и перенаправляем запрос на него через http
- ?? почему бы сразу не направить на нужный хост, если мы знаем ключ и хэш функцию
- Note also that the Orders Service is partitioned over three nodes, so GET requests must be routed to the correct node to get a certain key. This is handled automatically using the Interactive Queries functionality in Kafka Streams, although the example has to implement code to expose the HTTP endpoint.

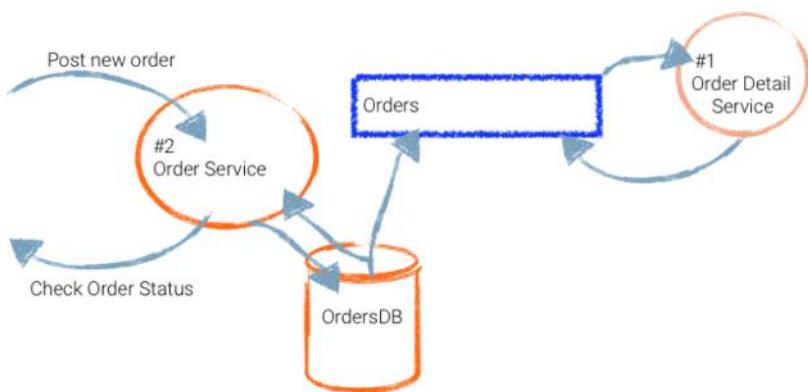
Option #1: Use a database too

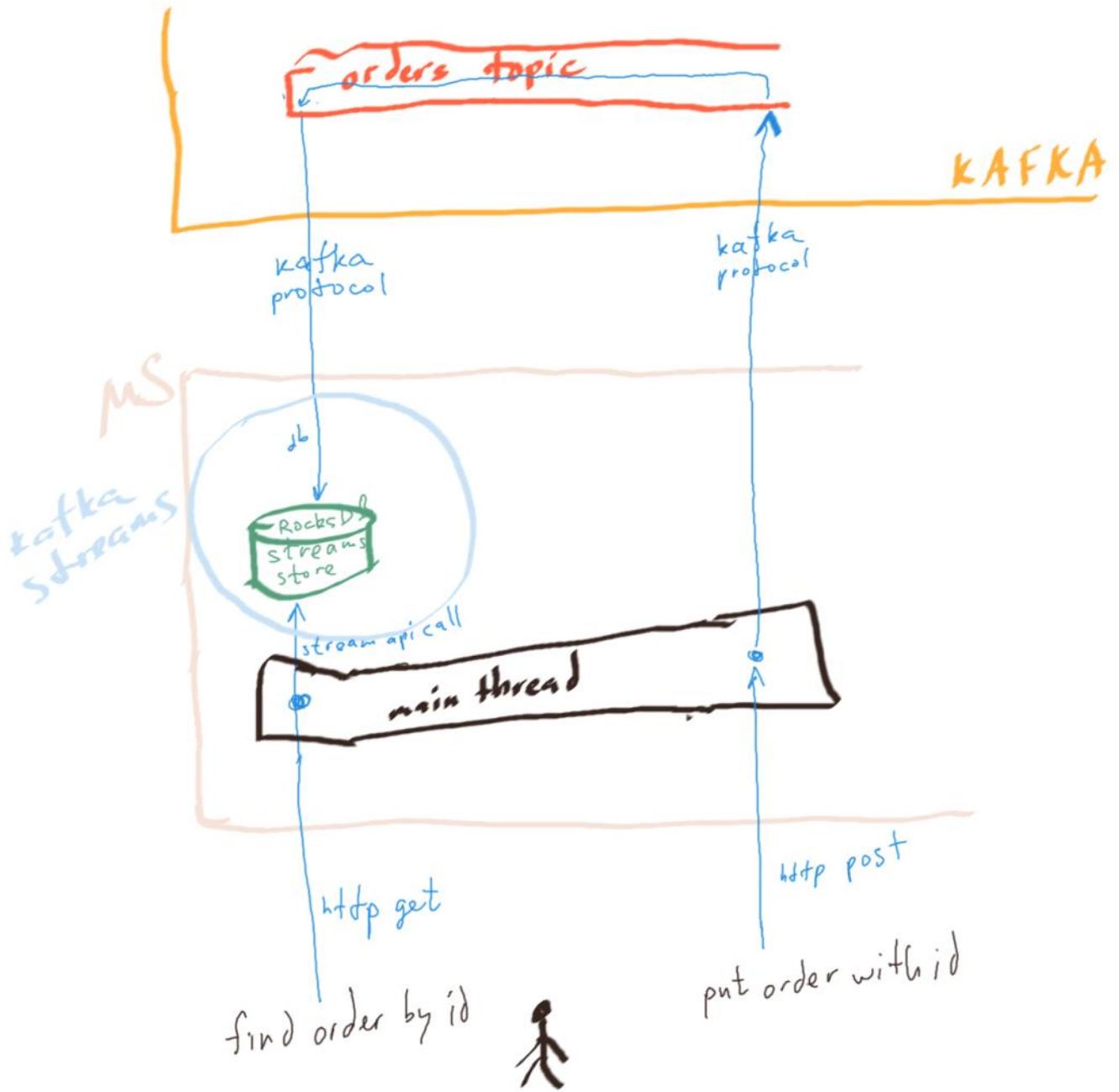


Option #2: Turn the DB inside out



Option 3 – Change Capture





Orders Webservice

1. POST /v1/orders <- Create new order
2. GET /v1/orders/<ID> <- Get order details and status

используется long polling

What's with the long poll?

```
private void maybeCompleteLongPollGet(String id, Order
order) {
    FilteredResponse callback = outstandingRequests.get(id);

    if (callback != null &&
        callback.predicate.test(id, order)) {
        callback.asyncResponse.resume(order);
    }
}
```

стримсы использую не для трансформаций, а просто для того чтобы читать из топика и класть в локальную БД

Then we start the Stream

```
private KafkaStreams startKStreams(String configFile,
String stateDir) throws IOException {
    KafkaStreams streams = new KafkaStreams(
        createOrdersMaterializedView().build(),
        configStreams(configFile, stateDir,
        SERVICE_APP_ID));
    streams.start();
    return streams;
}
```

для того чтобы читать из локальной БД у kafka streams есть отдельный key-value API

How do we use this table?

```
private ReadOnlyKeyValueStore<String, Order> ordersStore() {
    return streams.store(
        ORDERS_STORE_NAME,
        QueryableStoreTypes.keyValueStore());
}
```

просто ищем в state store заказ и возвращаем по http

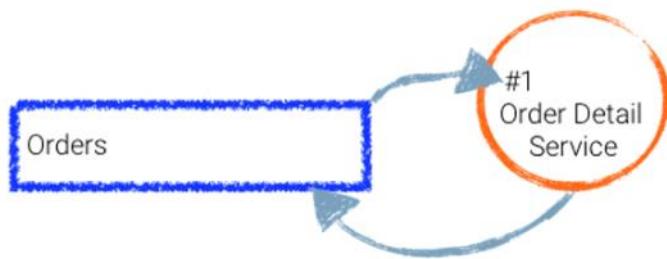
```
Order order = ordersStore().get(id);

if (order == null) {
    log.info("Delaying GET. Order not present for id " + id);
    outstandingRequests.put(id,
        new FilteredResponse<>(asyncResponse, (k, v) -> true));
} else {
    asyncResponse.resume(order);
}
```

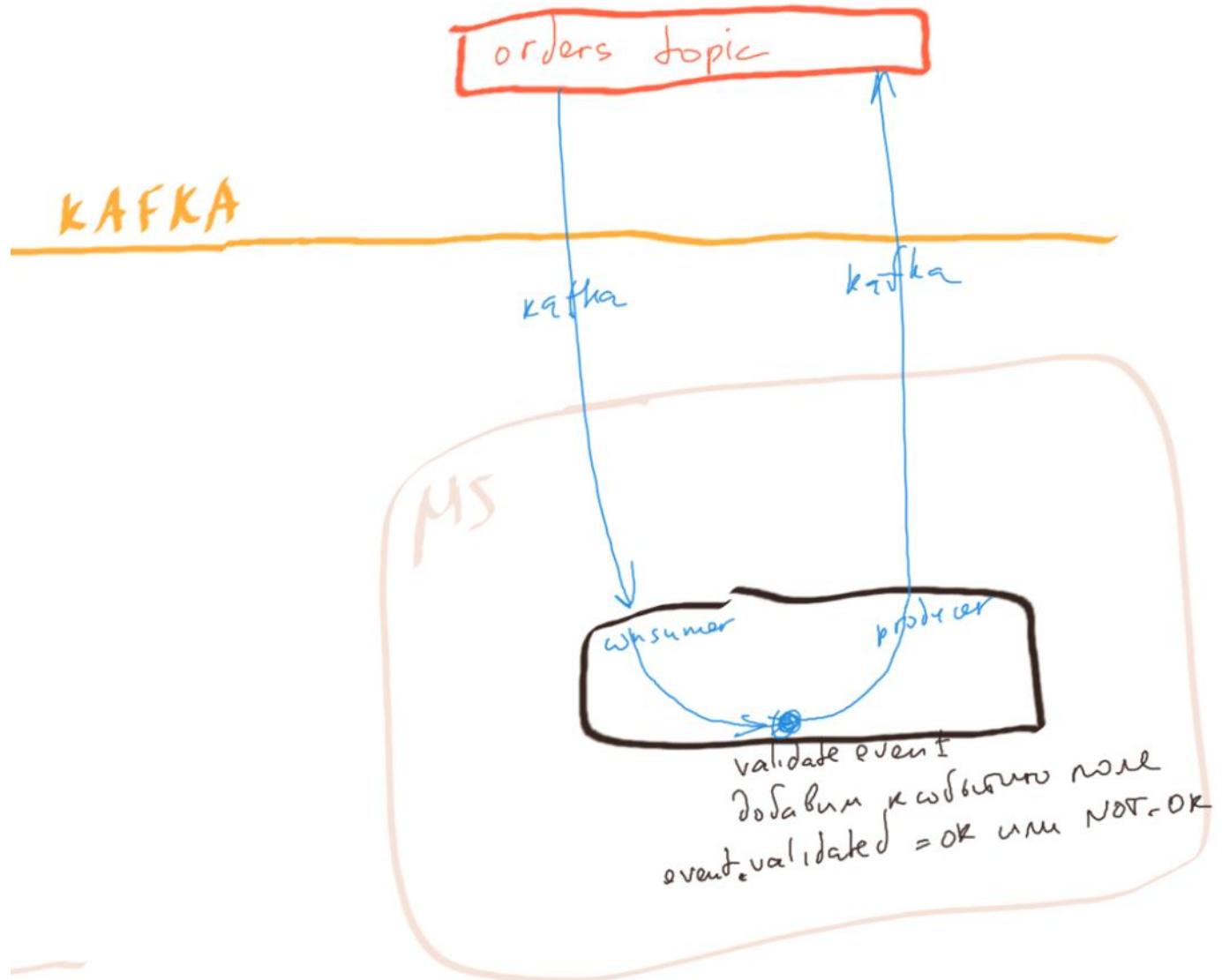
(1) сервис валидации просто выграбает сообщения из очереди "orders" и туда же кладет обратно то же сообщение но со статусом state=VALIDATED или state=FAILED

- можно добавить EOS

Step 1 – Validate Orders

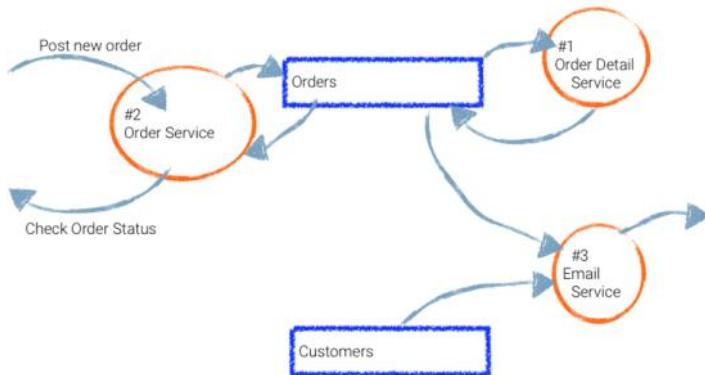


- Consume an Order event
- Check event:
 - Is there a product?
 - Is there a price?
 - Is the quantity positive?
 -
- Produce result:
Order event with status "valid"



(3) сервис email уведомления юзеров

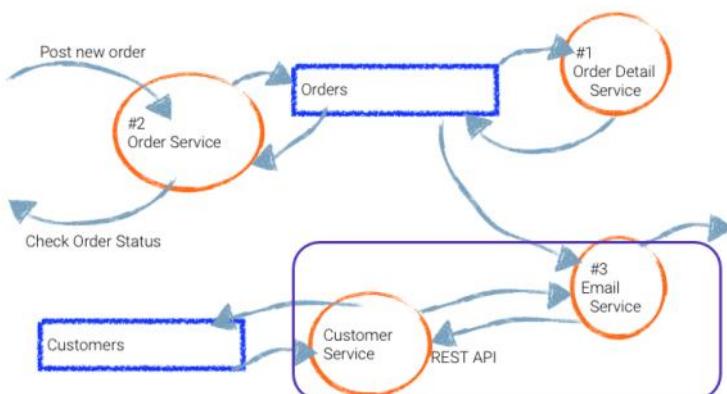
Maybe: Step 3 – Email Customers



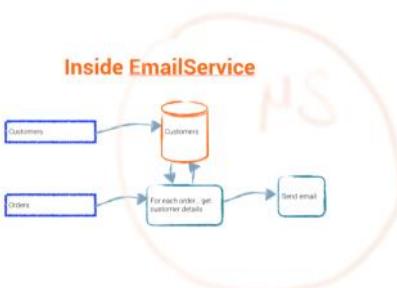
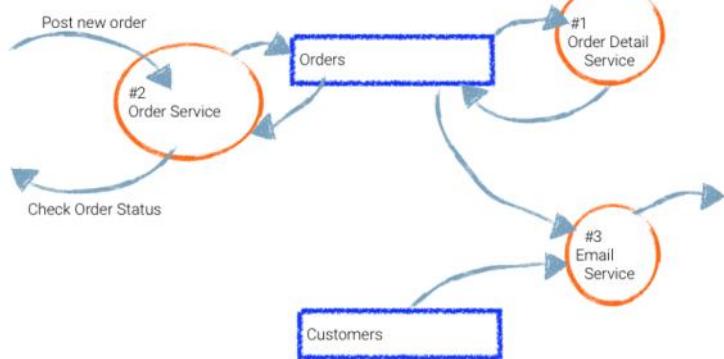
- Every time an order is created or updated...

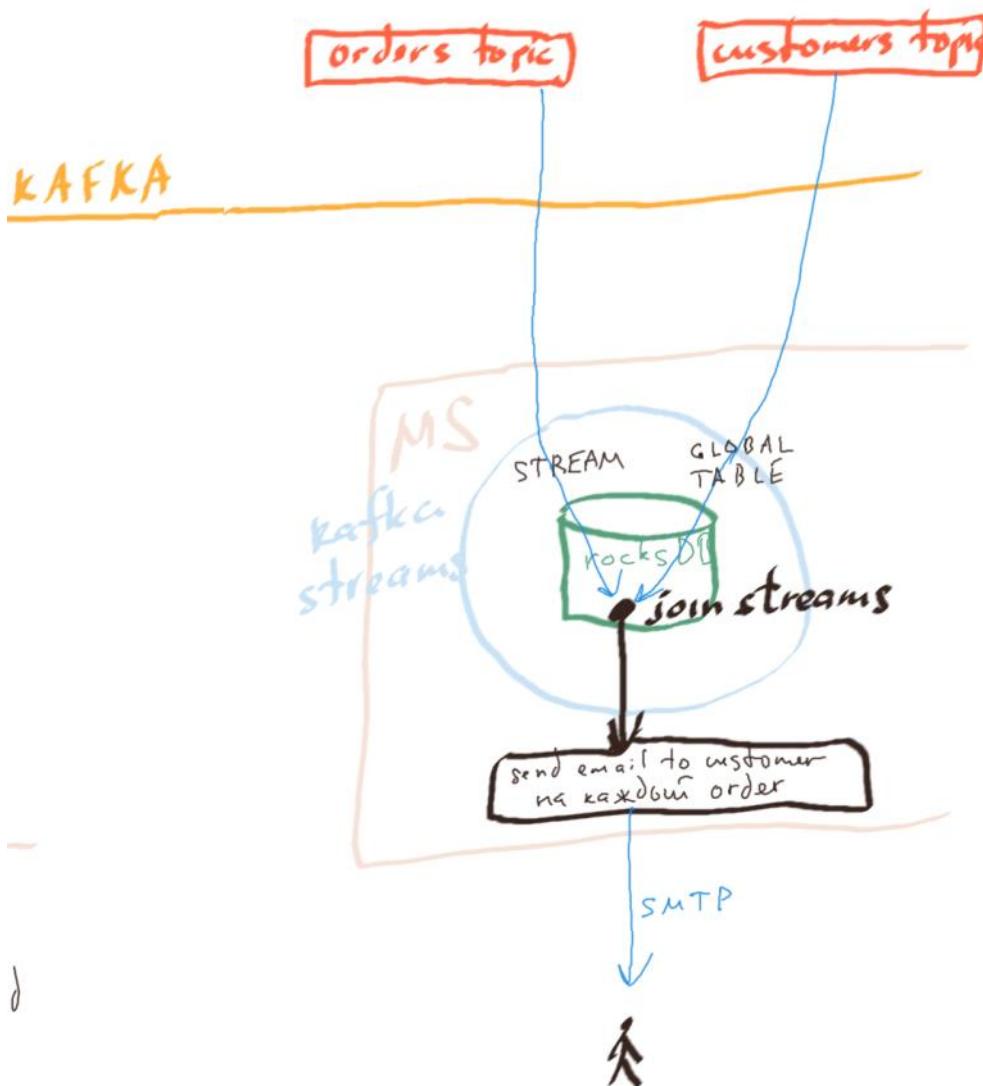
- Get order event
- Find the customer email
- Email customer with update

Option #1: Ask the customer service



Option #2: Create local view from stream of events





```
//Join customers and orders
orders.join(customers, Tuple::new)
//Consider confirmed orders for platinum customers
.filter((k, tuple) -> tuple.customer.level().equals(PLATINUM) && tuple.order.state().equals(CONFIRMED))
//Send email for each customer/order pair
.peek((k, tuple) ->emailer.sendMail(tuple));
```

вариант 2) алтернативно можно сделать join на KSQL а только последнюю операцию отправки почты на консьюмере

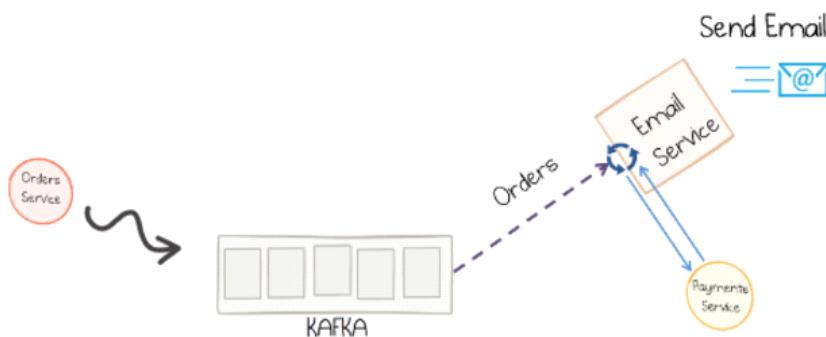
```
//Execute query in KSQL sidecar to filter stream
ksql> CREATE STREAM orders (ORDERID string, ORDERTIME bigint...) WITH (kafka_topic='orders', value_format='JSON');
ksql> CREATE STREAM platinum_emails AS
    select *
    from orders, customers
    where client_level == 'PLATINUM' and state == 'CONFIRMED';

//In Node.js service send Email
var nodemailer = require('nodemailer');
...
var kafka = require('kafka-node'),
Consumer = kafka.Consumer,
client = new kafka.Client(),
consumer = new Consumer(client, [ { topic: 'platinum_emails', partition: 0 } ] );
consumer.on('message', function (orderConsumerTuple) {
```

```
sendMail(orderConsumerTuple);
});
```

option1) event-driven approach

- Емейл сервис может реагировать на события заказа, а затем искать соответствующую оплату. Или он может сделать обратное: отреагировать на платежи, а затем найти соответствующий заказ. Допустим, первое



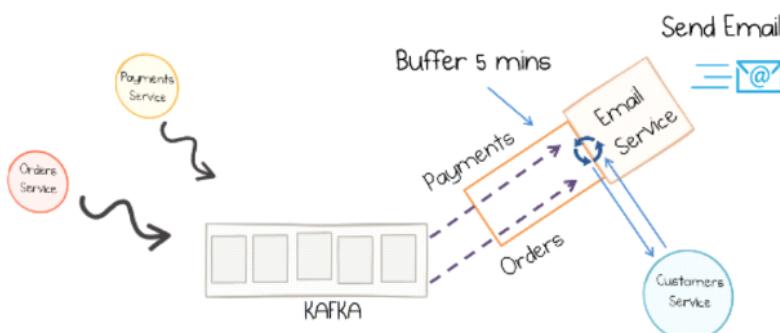
проблемы подхода

- Постоянная потребность искать информацию, одно сообщение за раз.
- Платеж и заказ создаются примерно одновременно, поэтому один может прийти раньше другого. Это означает, что если заказ поступит в службу электронной почты до того, как платеж будет доступен в базе данных, то нам придется либо блокировать и опрашивать, пока он не станет доступен, либо, что еще хуже, полностью пропустить обработку электронной почты.

option2) Pure (Stateless) Streaming Approach

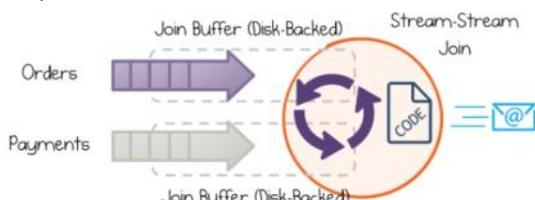
stream+stream join

- Потоки буферизуются до тех пор, пока не появятся оба события, и их можно объединить
- Это решает две вышеупомянутые проблемы с Option1(см выше). Нет удаленного поиска, обращающегося к первой точке. Также больше не имеет значения, в каком порядке прибывают события.
- Когда Kafka Streams перезапускается, перед выполнением какой-либо обработки он перезагружает содержимое каждого буфера (и может быть ошибка)



Мы применяем соединение поток-поток, которое ожидает появления соответствующих событий заказа и платежа в службе электронной почты перед запуском кода электронной почты. Соединение во многом похоже на логическое И.

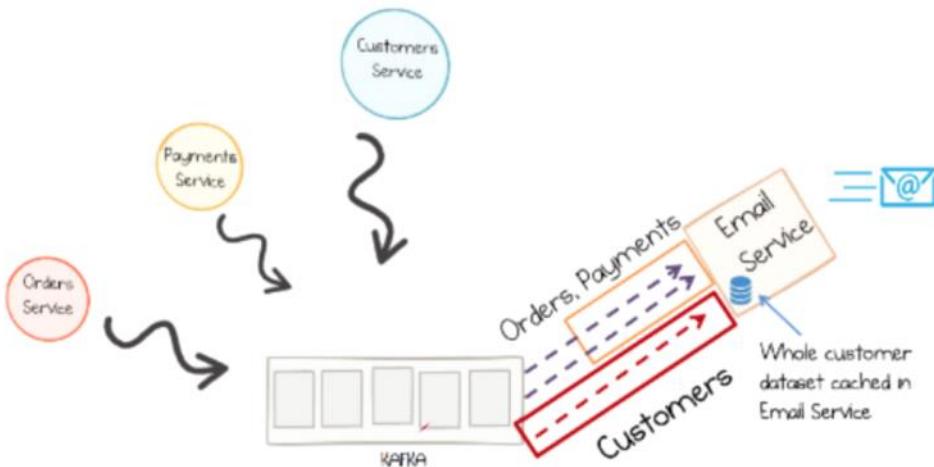
Таким образом, независимо от того, какое событие появится позже, соответствующее событие можно быстро извлечь из буфера, чтобы операция соединения могла завершиться.



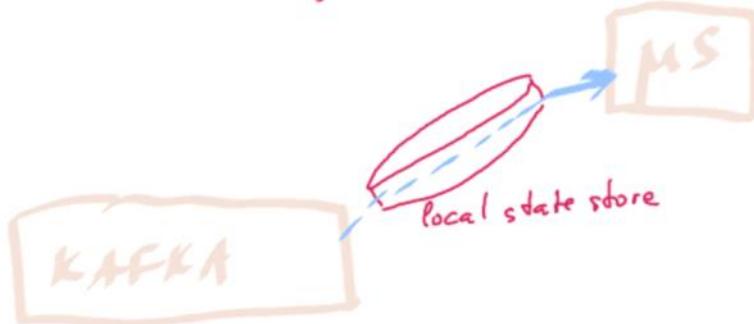
option3) The Stateful Streaming Approach

stream+table join

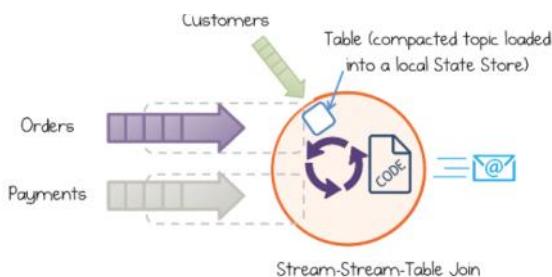
- Он использует тот же процесс локальной буферизации, который используется для обработки потенциальных задержек в темах заказов и платежей (см Option2 выше), но вместо буферизации всего на несколько минут он предварительно загружает весь поток событий клиентов из Kafka в службу электронной почты, где его можно использовать для найдите исторические значения (рис. 6-4).
- Служба потоковой передачи с отслеживанием состояния, которая реплицирует тему Customers-клиентов в локальную таблицу, хранящуюся в API Kafka Streams
- .



Интересный способ изображения local state store



Хорошая вещь в использовании таблицы состоит в том, что она ведет себя так же, как таблицы в базе данных. Поэтому, когда мы присоединяем поток заказов к таблице клиентов, нам не нужно беспокоиться о сроках хранения, окнах или других подобных сложностях. На рисунке 14-4 показано трехстороннее соединение между заказами, платежами и клиентами, где клиенты представлены в виде таблицы.



недостатки подхода

- Теперь служба отслеживает состояние, что означает, что для работы экземпляра службы электронной почты требуются соответствующие данные клиента. В худшем случае это означает загрузку полного набора данных при запуске.

преимущества подхода

- Услуга больше не зависит от наихудшей производительности или живучести службы поддержки клиентов.
- Сервис может обрабатывать события быстрее, поскольку каждая операция выполняется без сетевого вызова.
- Сервис может выполнять больше операций, ориентированных на данные, с данными, которые он хранит. Этот заключительный момент особенно важен для систем, все более ориентированных на данные, которые мы строим сегодня. В качестве примера представьте, что у нас есть графический интерфейс, который позволяет пользователям просматривать информацию о заказах, платежах и клиентах в прокручиваемой сетке. Сетка позволяет пользователю прокручивать вверх и вниз отображаемые элементы. В традиционной модели без сохранения состояния каждая строка на экране требует вызова всех трех служб. На практике это было бы вялым, поэтому, вероятно, будет добавлено кеширование вместе с неким вручную созданным механизмом опроса для поддержания кеша в актуальном состоянии.

Join-Filter-Process паттерн

Этот шаблон наблюдается в большинстве служб, но, вероятно, лучше всего демонстрируется службой электронной почты, которая объединяет заказы, платежи и клиентов, пересылая результат в функцию, которая отправляет электронное письмо.

1. Join. The DSL is used to join a set of streams and tables emitted by other services.
2. Filter. Anything that isn't required is filtered. Aggregations are often used here too.
3. Process. The join result is passed to a function where business logic executes. The output of this business logic is pushed into another stream.

пример BIG

20 декабря 2020 г. 21:55

<https://github.com/confluentinc/examples/tree/6.0.1-post/microservices-orders>

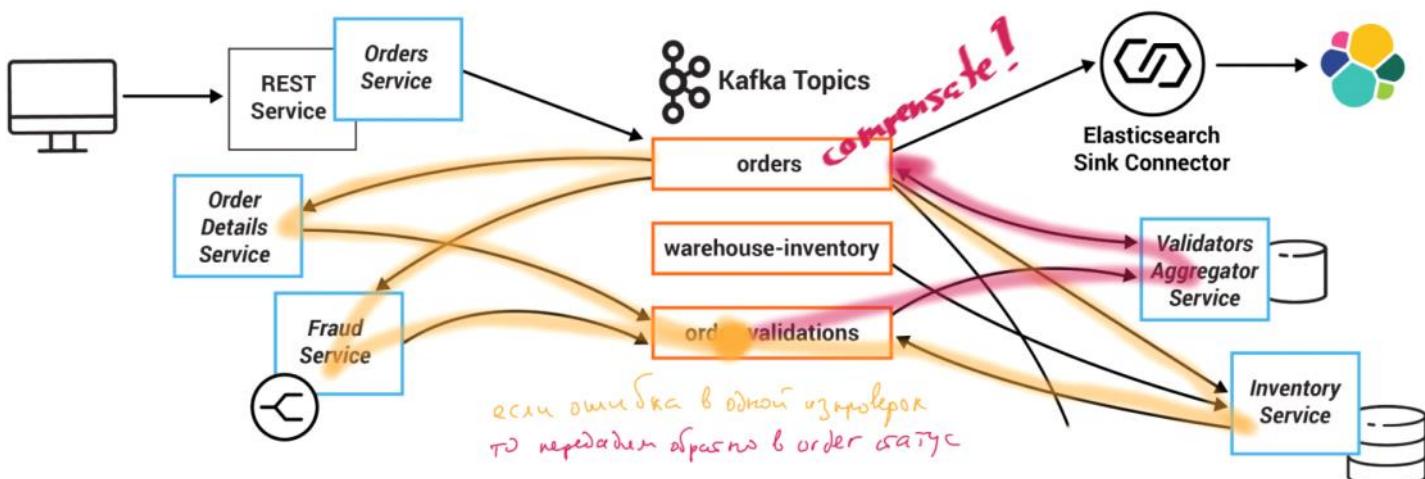
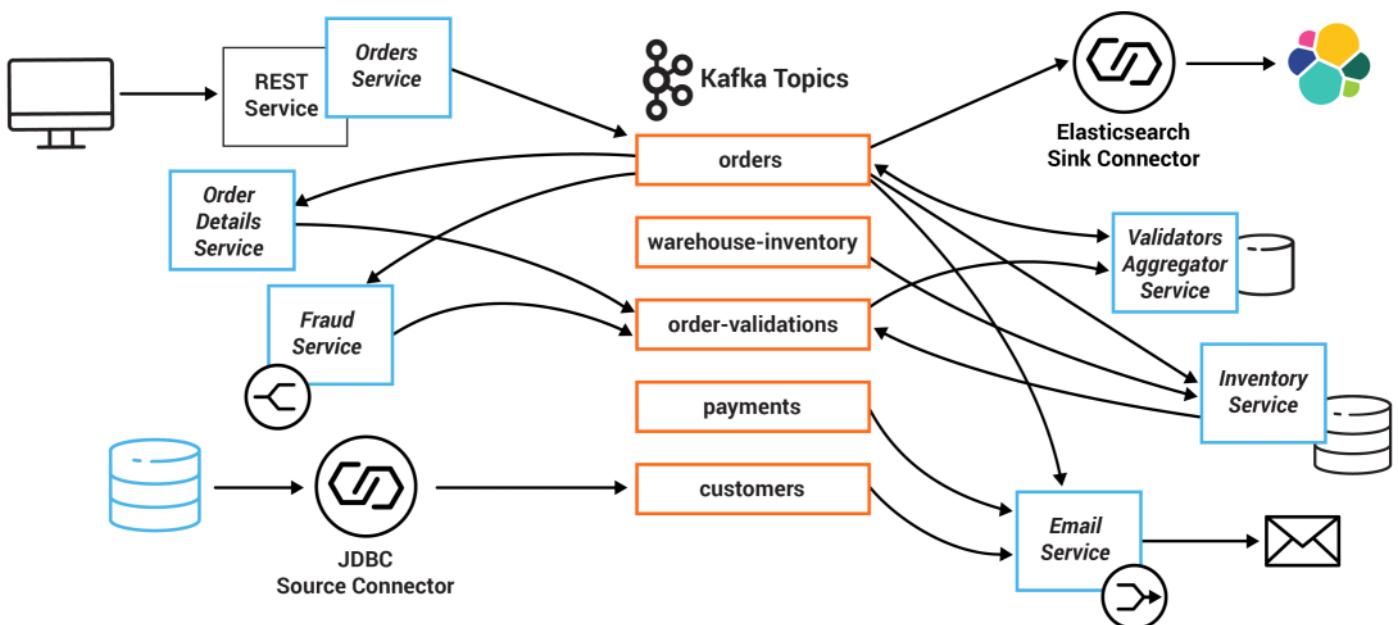
<https://github.com/confluentinc/kafka-streams-examples>

<https://github.com/confluentinc/kafka-streams-examples/tree/6.0.1-post/src/main/java/io/confluent/examplesstreams/microservices>

<https://docs.confluent.io/platform/current/tutorials/examples/microservices-orders/docs/index.html>

<https://www.confluent.io/blog/building-a-microservices-ecosystem-with-kafka-streams-and-ksql/>

<https://www.confluent.io/blog/turning-the-database-inside-out-with-apache-samza/>



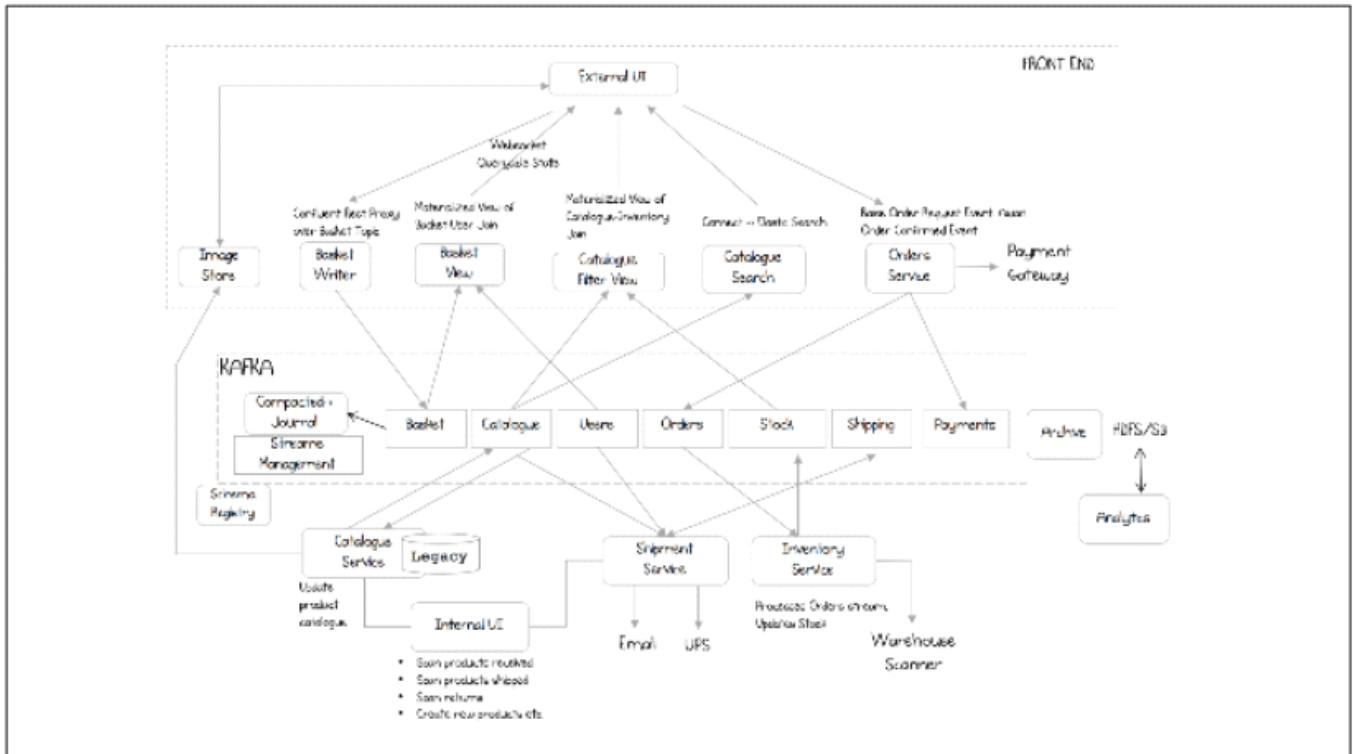


Figure 15-8. A more holistic streaming ecosystem

Basket writer + Basket view [не отдельный топик для представления в виде корзины]

These represent an implementation of CQRS, as discussed in “Command Query Responsibility Segregation” on page 61 in Chapter 7. The Basket writer proxies HTTP requests, forwarding them to the Basket topic in Kafka when a user adds a new item. The Confluent REST proxy (which ships with the Confluent distribution of Kafka) is used for this. The Basket view is an event-sourced view, implemented in Kafka Streams, with the contents of its state stores exposed over a REST interface in a manner similar to the orders service in the example discussed earlier in this chapter (Kafka Connect and a database could be substituted also). The view represents a join between User and Basket topics, but much of the information is thrown away, retaining only the bare minimum: `userId → List[product]`. This minimizes the view’s footprint.

The Catalogue Filter view

This is another event-sourced view but requires richer support for pagination, so the implementation uses Kafka Connect and Cassandra.

Catalogue search

A third event-sourced view; this one uses Solr for its full-text search capabilities.

Orders service (+payment gateway)

Orders are validated and saved to Kafka. This could be implemented either as a single service or a small ecosystem like the one detailed earlier in this chapter.

Catalog service

A legacy codebase that manages changes made to the product catalog, initiated from an internal UI. This has comparatively fewer users, and an existing codebase. Events are picked up from the legacy Postgres database using a CDC connector to push them into Kafka. The single-message transforms feature reformats the messages before they are made public. Images are saved to a distributed filesystem for access by the web tier.

Shipping service (+email +UPS)

A streaming service leveraging the Kafka Streams API. This service reacts to orders as they are created, updating the Shipping topic as notifications are received from the delivery company.

Inventory service

Another streaming service leveraging the Kafka Streams API. This service updates inventory levels as products enter and leave the warehouse.

Archive

All events are archived to HDFS, including two, fixed T-1 and T-10 point-in-time snapshots for recovery purposes. This uses Kafka Connect and its HDFS connector.

Streams management

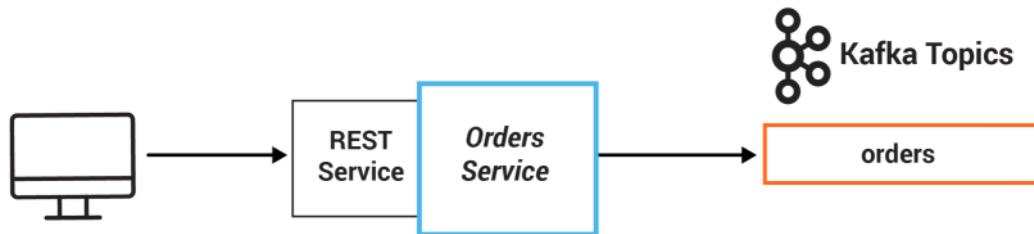
A set of stream processors manages creating latest/versioned topics where relevant (see the Latest-Versioned pattern in “Long-Term Data Storage” on page 25 in Chapter 3). This layer also manages the swing topics used when non-backward-compatible schema changes need to be rolled out. (See “Handling Schema Change and Breaking Backward Compatibility” on page 124 in Chapter 13.)

Schema Registry

The Confluent Schema Registry provides runtime validation of schemas and their compatibility.

... Exercise 1 (-""-)

20 декабря 2020 г. 22:38

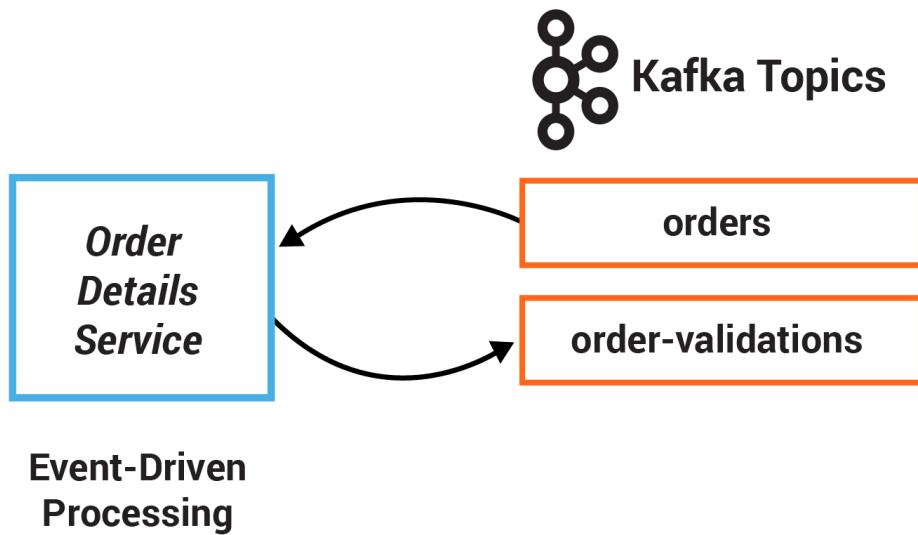


возможно потребуется поиск другого стримс-клиента если данного ключа в данной партиции не оказалось

- тогда через кафка-метаданные ищем хост и перенаправляем запрос на него через http
- ?? почему бы сразу не направить на нужный хост, если мы знаем ключ и хэш функцию

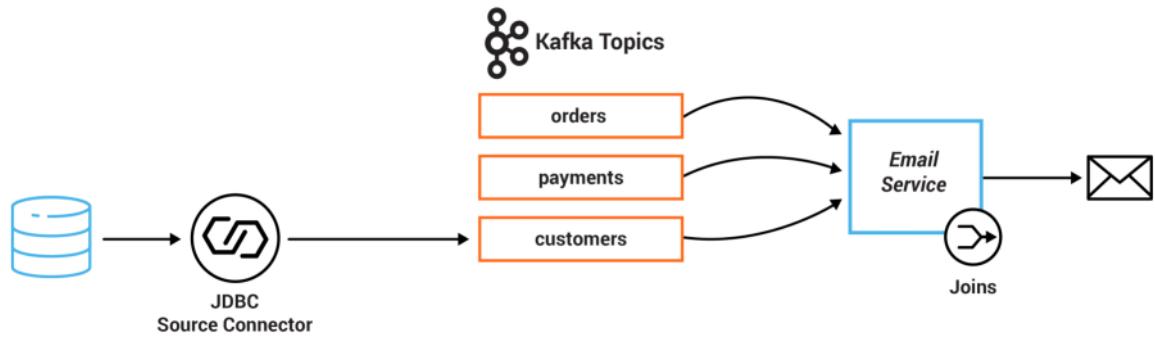
... Exercise 2 (-"-")

20 декабря 2020 г. 22:38



... Exercise 3 (-"-")

20 декабря 2020 г. 23:18

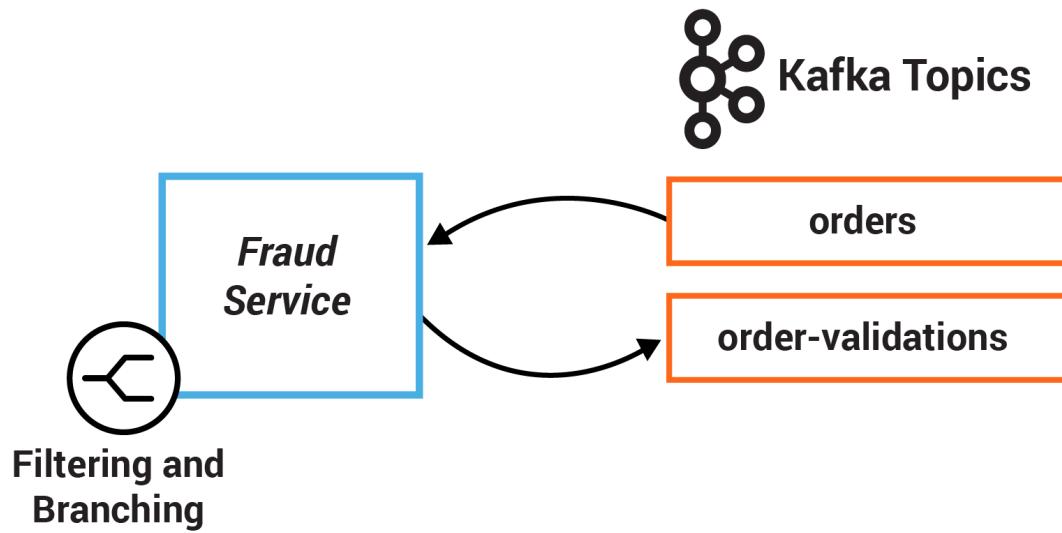


First, the payment stream needs to be rekeyed to match the same key info as the order stream before joined together.

еще дополнительно можно отослать в топик с соответствующим уровнем обслуживания клиента

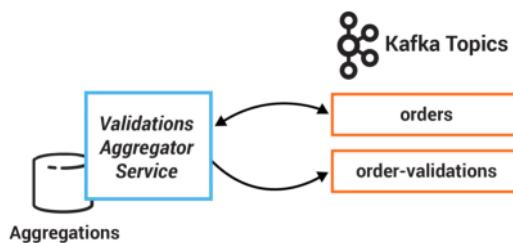
... Exercise 4

20 декабря 2020 г. 23:18



... Exercise 5

20 декабря 2020 г. 23:18



Aggregations

- Три механизма проверки (мошенничество, инвентаризация, детали заказа) подписываются на эти события и выполняются параллельно, выдавая PASS или FAIL в зависимости от успешности каждой проверки. Результат этих проверок проходит через отдельную тему, Looking at the Orders Service first, a REST interface provides methods to POST and GET Orders. Posting an Order creates an event in Kafka. This is picked up by **three different validation engines** (Fraud Check, Inventory Check, Order Details Check) which validate the order in parallel, emitting a PASS or FAIL based on whether each validation succeeds. **The result of each validation is pushed through a separate topic, Order Validations, so that we retain the 'single writer' status of the Orders Service —> Orders Topic.** The results of the various validation checks are aggregated back in the Order Service (Validation Aggregator) which then moves the order to a Validated or Failed state, based on the combined result. This is essentially an implementation of the Scatter-Gather design pattern.
- Глядя на заказы на обслуживание первых, интерфейс REST предоставляет методы POST и GET Орденов. Размещение заказа создает событие в Kafka. Это подобрано три различных движателями проверки (Каталог мошенничества, инвентаризации Проверить, Деталь заказа Проверить), который Validate порядок параллельно, исключающим или несоответствие, основываясь на успешности каждой проверки. Результат каждой проверки проходит через отдельную тему, «Проверка заказов», так что мы сохраним статус «единого писателя» Службы заказов -> Тема заказов. Результаты различных проверок достоверности собираются обратно в Службе заказов (агрегатор проверки), который затем переводит заказ в состояние Validated или Failed на основе комбинированного результата. По сути, это реализация шаблона проектирования Scatter-Gather .

```
private KafkaStreams aggregateOrderValidations( final String bootstrapServers,
                                                final String stateDir,
                                                final Properties defaultConfig )
{
    final int numberOfRules = 3; //TODO put into a KTable to make dynamically configurable

    final StreamsBuilder builder = new StreamsBuilder();
    final KStream<String,OrderValidation> validations = builder.stream( ORDER_VALIDATIONS.name(), serde1 );
    final KStream<String,Order> orders = builder.stream( ORDERS.name(), serde2 )
        .filter( ( id, order ) -> OrderState.CREATED.equals( order.getState() ) );

    //This code goes from OrderValidation to order
    validations.groupByKey( serde3 )
        .windowedBy( SessionWindows.with( Duration.ofMinutes( 5 ) ) )
        .aggregate( () -> OL, ( id, result, total ) -> PASS.equals( result.getValidationResult() ) ? total + 1 : total, ( k, a, b ) -> b == null ? a : b, //include a merger as we're using session windows.
        Materialized.with( null, Serdes.Long() ) )
    //get rid of window
    .toStream( ( windowedKey, total ) -> windowedKey.key() )
    //When elements are evicted from a session window they create delete events. Filter these.
    .filter( ( k1, v ) -> v != null )
    //only include results were all rules passed validation
    .filter( ( k, total ) -> total <= numberOfRules )
    //Join back to orders
    .join( orders, ( id, order ) ->
        //Set the order to Validated
        newBuilder( order ).setState( VALIDATED )
            .build(), JoinWindows.of( Duration.ofMinutes( 5 ) ), serde4 )
    //Push the validated order into the orders topic
    .to( ORDERS.name(), serde5 );

    //This code goes from order to OrderValidation
    validations.filter( ( id, rule ) -> FAIL.equals( rule.getValidationResult() ) )
        .join( orders, ( id, order ) ->
            //Set the order to Failed and bump the version on its ID
            newBuilder( order ).setState( OrderState.FAILED )
                .build(), JoinWindows.of( Duration.ofMinutes( 5 ) ), serde6 )
    //there could be multiple failed rules for each order so collapse to a single order
    .groupByKey( serde6 )
    .reduce( ( order, v1 ) -> order )
    //Push the validated order into the orders topic
    .toStream()
    .to( ORDERS.name(), Produced.with( ORDERS.keySerde(), ORDERS.valueSerde() ) );
}

return new KafkaStreams( builder.build(), baseStreamsConfig( bootstrapServers, stateDir, SERVICE_APP_ID, defaultConfig ) );
}
```

scatter-gather pattern

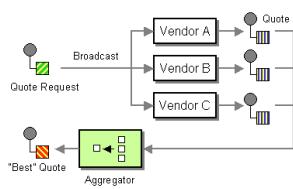
The result of each validation is pushed through a separate topic, Order Validations, so that we retain the 'single writer' status of the Orders Service —> Orders Topic. The results of the various validation checks are aggregated back in the Order Service (Validation Aggregator) which then moves the order to a Validated or Failed state, based on the combined result. This is essentially an implementation of the Scatter-Gather design pattern.

- Результаты различных проверочных проверок собираются обратно в Службе заказов (агрегатор проверки), который затем переводит заказ в состояние Validated или Failed на основе комбинированного результата. По сути, это реализация шаблона проектирования Scatter-Gather .

<https://www.enterpriseintegrationpatterns.com/patterns/messaging/BroadcastAggregate.html>

В примере обработки заказа, представленном в предыдущих шаблонах, каждая позиция заказа, которая в настоящее время отсутствует на складе, может быть поставлена одним из нескольких внешних поставщиков. Однако поставщики могут иметь или не иметь соответствующий товар на складе сами, они могут назначать другую цену и могут иметь возможность поставить деталь к другой дате. Чтобы выполнить заказ наилучшим образом, мы должны запросить расценки у всех поставщиков и решить, какой из них предоставит нам лучший срок для запрашиваемого товара.

Как вы поддерживаете общий поток сообщений, когда сообщение нужно отправить нескольким получателям, каждый из которых может отправить ответ?



Используйте *Scatter-Gather*, который рассыпает сообщение нескольким получателям и повторно объединяет ответы в одно сообщение.

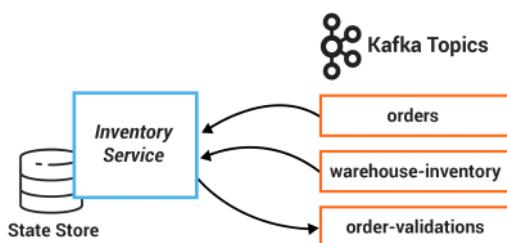
Scatter-Gather маршрутизирует сообщение с запросом к нескольким получателям. Затем он использует [аггрегатор](#) для сбора ответов и преобразования их в одно ответное сообщение.

паттерн: [Scatter-Gather](#)

- относительно распространенный вариант использования в бизнес-системах - ожидание наступления N событий. Это тривиально, если каждое событие находится в своей отдельной теме - это просто трехстороннее соединение, - но если все события прибывают по одной теме, это требует немного большего размышления.
 - Служба заказов в примере, рассмотренном ранее в этой главе (рис. 15-1), ожидает результатов проверки от каждой из трех служб проверки, отправленных через одну и ту же тему. В целом проверка успешна только в том случае, если все три возвращают PASS.
- Предполагая, что вы подсчитываете сообщения с определенным ключом, решение принимает форму
1. Group by the key.
 2. Count occurrences of each key (using an aggregator executed with a window).
 3. Filter the output for the required count.

... Exercise 6 (state store)

20 декабря 2020 г. 23:18



партitionирование по product id

1. Kafka's transactions feature is enabled.
2. Data is partitioned by ProductId before the reservation operation is performed.

- Разделение по ProductId немного сложнее. Разделение гарантирует, что все заказы для iPad отправляются в один поток в одном из доступных экземпляров службы, что гарантирует выполнение заказа. Заказы на другие товары будут отправлены в другое место. Таким образом, при распределенном развертывании это гарантирует выполнение заказов на продукты одного и того же типа, iPad, iPhone и т. д., без необходимости межсетевой координации. (Подробнее см. в разделе «Масштабирование одновременных операций в потоковых системах» книги «Проектирование систем, управляемых событиями».)
- Partitioning by ProductId is a little more subtle. Partitioning ensures that all orders for iPads are sent to a single thread in one of the available service instances, guaranteeing in-order execution. Orders for other products will be sent elsewhere. So in a distributed deployment, this guarantees in-order execution for orders for the same type of product, iPads, iPhones, etc., without the need for cross-network coordination. (For more detail see the section “Scaling Concurrent Operations in Streaming Systems” in the book Designing Event-Driven Systems.)

rekey / repartitioning

ко-партиционирование - data arranged in this way is termed co-partitioned.

- Служба инвентаризации должна переупорядочить заказы так, чтобы они обрабатывались по ProductId . Это выполняется с помощью операции, называемой rekey , которая перемещает заказы в новую промежуточную тему в Kafka, на этот раз с ключом ProductId , а затем обратно в службу инвентаризации.

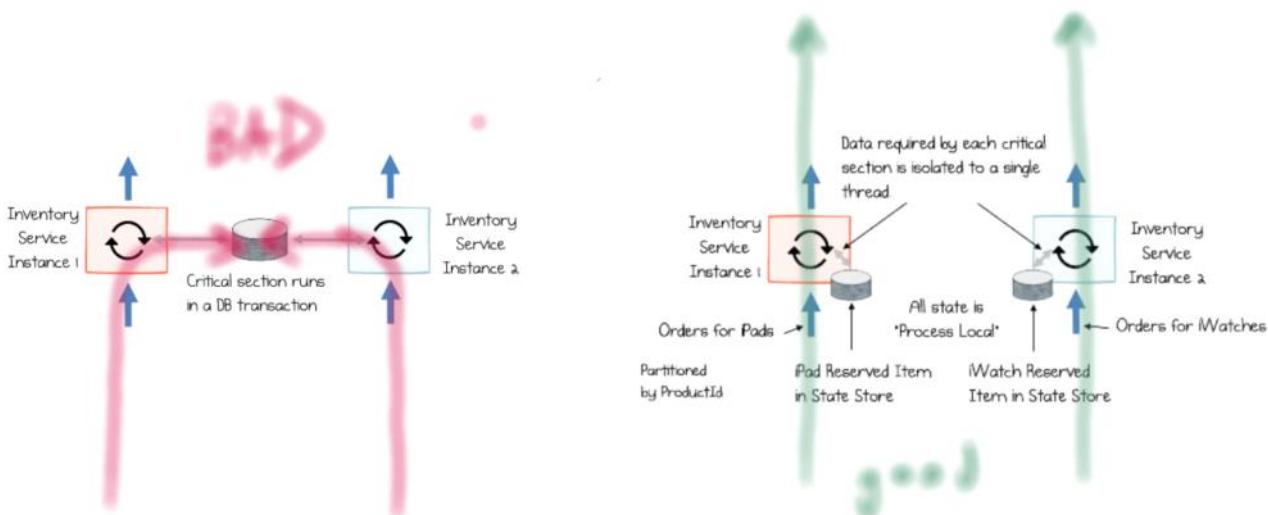
`orders.selectKey((id, order) -> order.getProduct())//rekey by ProductId`

после rekey мы можем сделать джойн по товару

- Once rekeyed, the join condition can be performed without any additional network access required. For example, inventory with productId=5 is collocated with orders for productId=5.

обеспечение параллелизма за счет партиционирования

- каждый инстанс микросервиса обрабатывает свой товар
- для того чтобы сделать join нужно сделать репартиционирование заказов по product_id



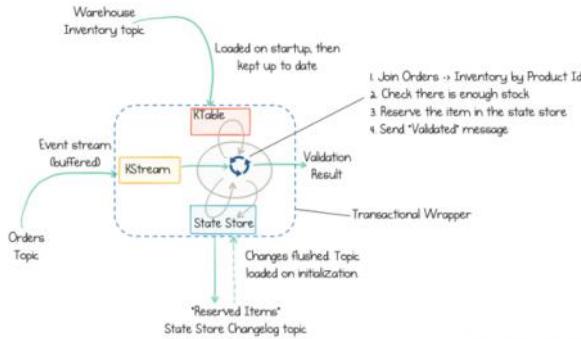
т.е "order microservice" может хранить во вьюхе состояние запасов, но чтобы зарезервировать товар (в соответствии с single writer principle) он должен обратиться в "inventory microservice"

- stream + table join
- т.е каждый инстанс микросервиса в своей локальной БД хранит свою шарду (состояние запасов по своему кусочку товаров, т.е все товары поделены по всем инстансам)
- все операции заключены в транзакцию

- для того чтобы сделать join нужно сделать репартиционирование заказов по `product_id`
- Сервис выполняет простую операцию: когда пользователь покупает iPad, он проверяет наличие достаточного количества iPad, доступных для выполнения заказа, затем физически резервирует некоторое количество из них, чтобы никакой другой процесс не мог их купить:

1. Read an Order message
2. Validate whether there is enough stock available for that Order: (iPads in the warehouse KTable) - (iPads reserved in the State Store).
3. Update the State Store of "reserved items" to reserve the iPad so no one else can take it.
4. Send out a message that Validates the order.

You can find the code for this service [here](#).



- The inventory service does this with a Kafka Streams state store (a local, disk-resident hash table, backed by a Kafka topic). So each thread executing will have a state store for "reserved stock" for some subset of the products. You can program with these state stores much like you would program with a hash map or key/value store, but with the benefit that all the data is persisted to Kafka and restored if the process restarts.

```
KeyValueStore<Product, Long> store = context.getStateStore(RESERVED);

//Get the current reserved stock for this product
Long reserved = store.get(order.getProduct());
//Add the quantity for this order and submit it back
store.put(order.getProduct(), reserved + order.getQuantity())
```

The keys used to partition the event streams must be invariant if ordering is to be guaranteed

правило-ограничение: ключи `ProductId` и `OrderId` в каждом заказе должны оставаться фиксированными во всех сообщениях, относящихся к этому заказу

Если я хочу удалить/добавить товар то нужно создать новый заказ

- Ключи, используемые для разделения потоков событий, должны быть неизменными, чтобы упорядочение было гарантировано. Таким образом, в данном конкретном случае это означает, что ключи `ProductId` и `OrderId` в каждом заказе приобретаемого товара, необходимо создать новый заказ)
- There are limitations to this approach, though. The keys used to partition the event streams must be invariant if ordering is to be guaranteed. So in this particular case it means the keys, `ProductId` and `OrderId`, on each order must remain fixed across all messages that relate to that order. Typically, this is a fairly easy thing to manage at a domain level (for example, by enforcing that, should a user want to change the product they are purchasing, a brand new order must be created).
- Это ограничение нужно чтобы данный подход джойна и тэгу работал

В примере: один order содержит только один product (а не массив)

- те как бы денормализованные данные
- событием считается выбор одного товара юзером (а не когда он собирает всю корзину)
- это находится в согласии с законом "journal the whole fact as it arrived" см "onenote:#%20объект%20эт%20полное-состояние%20или%20дельта§ion-id={712B8D4-BDE4-4FFF-A011-59DE79779A94}&page-id={53A24205-3590-4EC8-BBF7-97057F636F61}&end&base-path=C:\Users\trans\Osync\vova_from_onenote\lf_algonote_v1\SYSTEMDESIGN\KAFKA_EVENT%20DRIVEN.one"
 - что происходит, когда пользователь отменяет одну позицию? Простое решение - просто снова записать все это в журнал как еще один агрегированный, но отмененный. Но что, если по какой-то причине заказ недоступен, а все, что мы получаем, - это одна отмененная позиция?
 - Эмпирическое правило - записывать то, что вы получаете, поэтому, если поступает только одну позицию товара то запишите только ее одну. Процесс объединения со всеми остальными товарами в заказе можно выполнить потом по чтению из materialized view.
 - И наоборот, разбиение события на подсобытия по мере их появления часто не является хорошей практикой по тем же причинам.
- см также правило "для того чтобы все это заработало нужно собирать данные в реальном времени как только юзер кликает(как в реактивном программировании)" onenote: EVENT%20DRIVEN.one#EVENT%20DRIVEN§ion-id={712B8D4-BDE4-4FFF-A011-59DE79779A94}&page-id={1F7E4313-F7E6-4981-B2FA-C2E28EB79181}&end&base-path=C:\Users\trans\Osync\vova_from_onenote\lf_algonote_v1\SYSTEMDESIGN\KAFKA

```
[{"namespace": "io.confluent.examples.streams.avro.microservices",
"type": "enum",
"name": "OrderState",
"symbols": ["CREATED", "VALIDATED", "FAILED", "SHIPPED"]},
{"namespace": "io.confluent.examples.streams.avro.microservices",
"type": "enum",
"name": "Product",
"symbols": ["JUMPERS", "UNDERPANTS", "STOCKINGS"]},
{"namespace": "io.confluent.examples.streams.avro.microservices",
"type": "record",
"name": "Order",
"fields": [
    {"name": "id", "type": "string"},
    {"name": "customerId", "type": "Long"},
    {"name": "state", "type": "OrderState"},
    {"name": "product", "type": "product"},
    {"name": "quantity", "type": "int"},
    {"name": "price", "type": "double"}],
},
{"namespace": "io.confluent.examples.streams.avro.microservices",
"type": "record",
"name": "OrderValue",
"fields": [
    {"name": "order", "type": "Order"},
    {"name": "value", "type": "double"}]
}

public enum Product
{
    GUITARS,
    AMPS,
    BOOKS
}
```

```

public class Order
{
    String id;
    String customerId;
    OrderState state;
    Product product;
    int quantity;
    double price;
}

В итоге обрабатываем все позиции товаров в корзине по одному товару за раз

private KafkaStreams processStreams( final String bootstrapServers,
final String stateDir,
final Properties defaultConfig )
{
    //Latch onto instances of the orders and inventory topics
    final StreamsBuilder builder = new StreamsBuilder();
    final KStream<String,Order> orders = builder.stream( Topics.ORDERS.name(), Consumed.with( Topics.ORDERS.keySerde(), Topics.ORDERS.valueSerde() ) );
    final KTable<Product,Integer> warehouseInventory = builder.table( Topics.WAREHOUSE_INVENTORY.name(), Consumed.with( Topics.WAREHOUSE_INVENTORY.keySerde(), Topics.WAREHOUSE_INVENTORY.valueSerde() ) );
    //Create a store to reserve inventory whilst the order is processed.
    //This will be prepopulated from Kafka before the service starts processing
    final StoreBuilder<KeyValueStore<Product,Long>> reservedStock = Stores.keyValueStoreBuilder( Stores.persistentKeyValueStore( RESERVED_STOCK_STORE_NAME ), Topics.WAREHOUSE_INVENTORY.keySerde(), Serdes.Long() )
        .withLoggingEnabled( new HashMap<>() );
    builder.addStateStore( reservedStock );
    //First change orders stream by Product (so we can join with warehouse inventory)
    orders.selectKey( ( id, order ) -> order.getProductId() ) // у одного заказа только один продукт
    //Limit to newly created orders
    .filter( ( id, order ) -> OrderState.CREATED.equals( order.getState() ) )
    //Join Orders to Inventory so we can compare each order to its corresponding stock value
    .join( warehouseInventory, KeyValue::new, Joined.with( Topics.WAREHOUSE_INVENTORY.keySerde(), Topics.ORDERS.valueSerde(), Serdes.Integer() ) )
    //Validate the order based on how much stock we have both in the warehouse and locally 'reserved' stock
    .transform( InventoryValidator::new, RESERVED_STOCK_STORE_NAME )
    //Push the result into the Order Validations topic
    .to( Topics.ORDER_VALIDATIONS.name(), Produced.with( Topics.ORDER_VALIDATIONS.keySerde(), Topics.ORDER_VALIDATIONS.valueSerde() ) );
    return new KafkaStreams( builder.build(), MicroserviceUtils.baseStreamsConfig( bootstrapServers, stateDir, SERVICE_APP_ID, defaultConfig ) );
}

private static class InventoryValidator
    implements Transformer<Product,KeyValue<Order, Integer>,KeyValue<String,OrderValidation> {
    private KeyValueStore<Product,Long> reservedStocksStore;

    @Override
    @SuppressWarnings( "unchecked" )
    public void init( final ProcessorContext context ) {
        reservedStocksStore = (KeyValueStore<Product,Long>) context.getStateStore( RESERVED_STOCK_STORE_NAME );
    }

    @Override
    public KeyValue<String,OrderValidation> transform( final Product productId,
                                                       final KeyValue<Order, Integer> orderAndStock ) {
        //Process each order/inventory pair one at a time
        final OrderValidation validated;
        final Order order = orderAndStock.key;
        final Integer warehouseStockCount = orderAndStock.value;

        //Look up Locally 'reserved' stock from our state store
        Long reserved = reservedStocksStore.get( order.getProductId() );
        if( reserved == null ) {
            reserved = 0L;
        }

        //If there is enough stock available (considering both warehouse inventory and reserved stock) validate the order
        if( warehouseStockCount - reserved - order.getQuantity() >= 0 ) {
            //reserve the stock by adding it to the 'reserved' store
            reservedStocksStore.put( order.getProductId(), reserved + order.getQuantity() ); //но не обрабатывается отмена предыдущего заказа, чтобы восстановить состояние запасов
            //validate the order
            validated = new OrderValidation( order.getId(), INVENTORY_CHECK, PASS );
        }
        else {
            //fail the order
            validated = new OrderValidation( order.getId(), INVENTORY_CHECK, FAIL );
        }
        return KeyValue.pair( validated.getOrderId(), validated );
    }

    @Override
    public void close() {
    }
}

```

КОНТЕКСТ

9 февраля 2021 г. 20:02

пример (анализ контекста событий)

- Измерение контекста - это говорит нам, относится ли определение к контексту, временному , пространственному , ориентированному на состояние или сегментация , или это составной контекст, то есть составленный из других спецификаций контекста

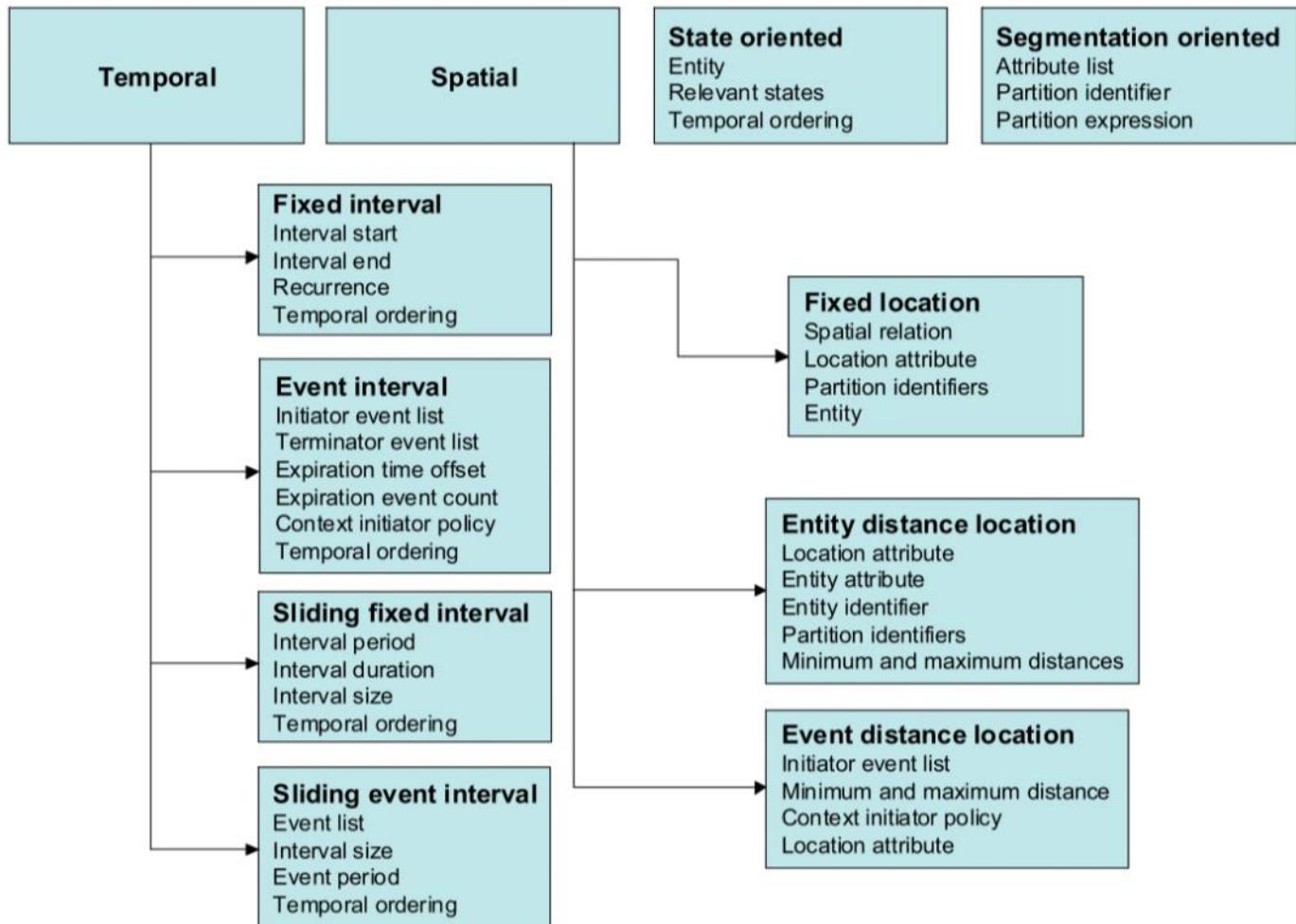


Figure 7.2 The different types of context showing their context parameters and partition parameters. This diagram also shows the principal dimension associated with a type, for example, the **fixed location** type is concerned with the spatial context dimension.

* temporal

9 февраля 2021 г. 20:03

- временной контекст состоит из одного или нескольких интервалов времени, возможнoperекрываются. Каждый временной интервал соответствует контекстному разделу, который содержит события, происходящие в течение этого интервала
- Агент обработки событий, который выдает предупреждение, если кто-то пытается сделать более трех снятий денег с банкомата в течение одного дня. В этом примере каждый день (начиная с полуночи) представляет собой отдельный контекстный раздел, содержащий события вывода средств, которые происходят в течение этого дня.

Когда агент обработки событий связан с временным контекстом, он обрабатывает только события, которые связаны с разделом этого контекста.

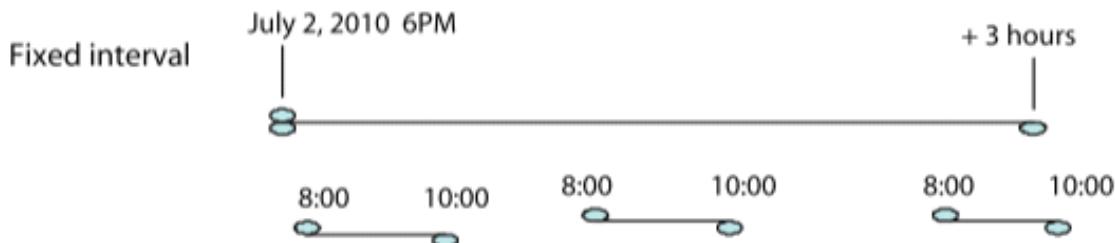
- разделы контекста разбивают входящий поток на один или несколько наборов событий, которые часто называют окнами.. Эти окна могут перекрываться, и в этом случае один экземпляр события может принадлежать более чем одному окну.

Each partition is defined by a time interval with a starting time and an ending time

- Есть несколько разных способов указать эти границы, и мы идентифицируем четыре разных типа временного контекста
- После того, как временной интервал установлен, раздел состоит из всех событий из соответствующего потока или потоков, которые лежат внутри этого интервала
- мы называем временные контекстные разделы окнами и используем фразы, открывающие окно или закрывающие окно, для обозначения того же, что и начало контекстного раздела или завершение контекстного раздела

(1) Fixed interval (задано абсолютное время)

- граница события определена как момент времени
- Фиксированный интервал контекста используется для представления либо одного периода времени фиксированной длины, или период времени фиксированной длины , которая повторяется в обычной манере, . Эти периоды времени не пересекаются.
- каждое окно представляет собой интервал, имеющий фиксированную продолжительность; может быть только одно окно или периодически повторяющаяся последовательность неперекрывающихся окон.



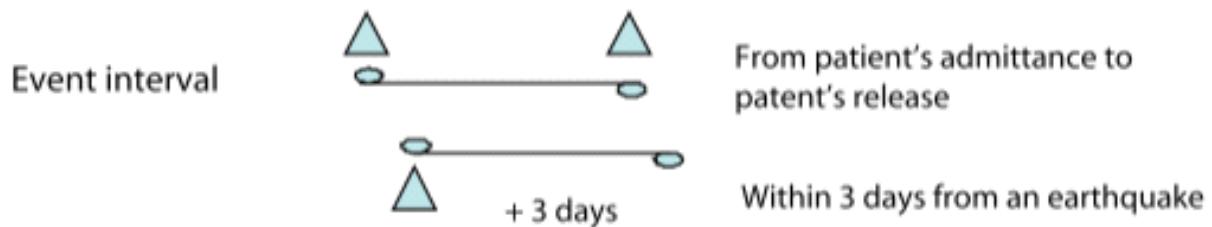
- *Interval start*—This gives the start time for the first (or only) interval. It can either be a DateTime or a Date. If a Date value is given by default, the interval

starts at midnight local time on the specified day, however, an application can set its own start of day time to apply to all fixed interval contexts.

- *Interval end*—This gives the end time for the first (or only) interval. It can be a Date or DateTime or it can be a duration giving the length of the interval. If it is specified as a Date, by default the interval ends at midnight that day; however, the application can specify an alternative time for end of day.
- *Recurrence*—This specifies how frequently the interval is to repeat, if at all.
- *Temporal ordering*—This parameter indicates whether the assignment of events to windows is based on their detection time or on their occurrence time timestamp. The event instance is only assigned to a window if it has the appropriate timestamp.

(2) Event interval

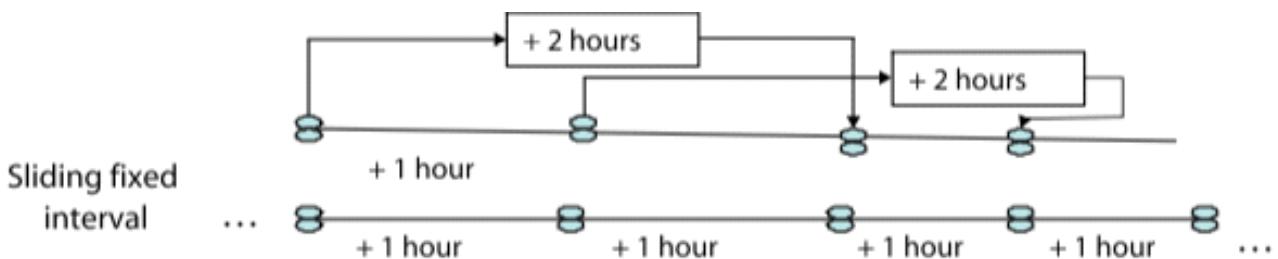
- граница определяется событием, нас интересуют события, которые происходят до или после этого события
 - окно открывается или закрывается, когда агент обработки событий получает конкретное событие в качестве входных данных.
 - Вы можете увидеть пример на рисунке 7.3, где есть окно, которое открывается событием приема пациента и закрывается событием освобождения пациента
 - каждое окно - это интервал, который начинается, когда связанный агент обработки событий получает событие, которое удовлетворяет заданному предикату. Он заканчивается, когда EPA получает событие, удовлетворяющее второму предикату, или по истечении заданного периода
-
- Событие терминатора не включается в окно, которое оно завершает.
 - Если событие одновременно является инициатором и завершателем, оно закрывает существующее окно перед открытием нового.
 - Событие, которое приводит к достижению лимита количества событий истечения срока, включается в окно (если только оно не является также завершающим событием).



- **Initiator event list**—A new window is opened when the event processing agent receives any of the events specified in the list. An event may be specified by its event type, in which case any instance of that event type will open the window, or it may be specified by the combination of an event type and a predicate expression. If the predicate is present, the window will be opened only if the event instance also satisfies the predicate (that is to say if the predicate expression returns TRUE when evaluated on the event instance).
- **Terminator event list**—The window is closed when the event processing agent receives any of the events specified in the terminator list. The terminator list is similar to the initiator list; each entry in the list consists of an event type and optionally a predicate expression. If the predicate expression is present, the window will only be closed if the event instance satisfies the predicate.
- **Expiration time offset**—The window is closed after this time period has elapsed, even if no terminator event has been received. By default this is an offset from the time the window was opened, but it can also be specified as an offset from any attribute of the initiator event whose data type is `DateTime`.
- **Expiration event count**—The window is closed after it has reached this size, even if no terminator event has been received.
- **Context initiator policy**—This parameter specifies what is to happen if a second initiator event is encountered. Context initiator policies are discussed in section 7.6.
- **Temporal ordering**—This parameter indicates whether inclusion of events in the window depends on the order in which they are detected, on their *occurrence time* timestamp, or on another specified attribute in the event that has a value that increases over time (a timestamp or an application-specified sequence number).

(3) Sliding fixed interval (время относительно предыдущего события)

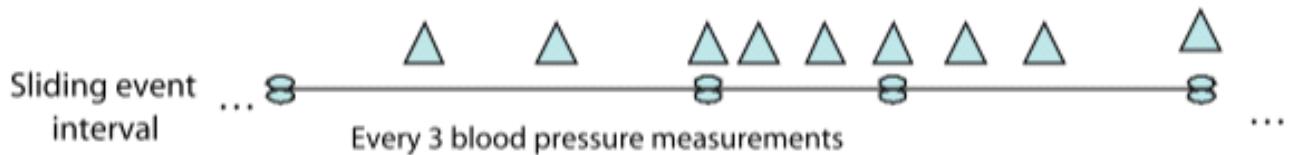
- интервалы могут пересекаться
- новые окна открываются через равные промежутки времени
- каждое окно открывается в указанное время после своего предшественника.
- Каждое окно имеет фиксированный размер, определяемый либо как временной интервал, либо как количество экземпляров событий.
- каждое окно представляет собой интервал с фиксированным временным размером или **фиксированным числом событий**.
- Новые окна открываются через равные промежутки времени относительно друг друга.
- Скользящие контексты с фиксированным интервалом обычно используются с агентами агрегирования.



- **Interval period**—The time period that elapses between the start of each window.
- **Interval duration**—The time period for which each window stays open.
- **Interval size**—The maximum number of event instances to be included in each window.
- **Temporal ordering**—This parameter indicates whether the assignment of events to windows is based on their detection time or on their occurrence time timestamp.

(4) Sliding event interval

- критерий открытия нового окна задается как количество событий, а не как период времени.
- На рисунке 7.3 вы можете увидеть пример, в котором каждая группа из трех последовательных измерений артериального давления назначается в новое окно, независимо от того, в какое время проводились измерения.
- открытие каждого нового окна и его продолжительность определяются путем подсчета количества событий, полученных агентом обработки событий
- подсчитывая количество раз, когда он получает определенные виды событий. Этот счетчик определяет, когда окна открываются и закрываются. Это непрерывный процесс, и можно иметь окна, расположенные вплотную друг за другом, или иметь перекрывающиеся окна.



- *Event list*—This specifies which events count towards the *interval size* and *event period*. The event processing agent may receive events not specified in this list; such events are included in the window but do not count towards the window size. Each entry in the list contains an event type. This type can be accompanied by a predicate expression, in which case an instance of the event type is only counted if it satisfies the predicate, that is to say, if the predicate returns TRUE when evaluated against the event instance.
- *Interval size*—This determines the size of each window. It is specified as the number of events (of a kind specified by the event list parameter) that are to be included in the window.
- *Event period*—The number of events (as specified by the event list parameter) received by the event processing agent before a new window is to be opened. If this parameter is not specified, it defaults to the interval size, which means that a new window is opened each time the previous window closes.
- *Temporal ordering*—This parameter indicates whether inclusion of events in the window depends on the order in which they are detected, on their *occurrence time*

* Spatial

9 февраля 2021 г. 20:04

- Пространственный контекст группирует экземпляры событий в соответствии с их геопространственными характеристиками
- предполагает, что событие содержит атрибут, который назначает местоположение событию.
- Точное значение этого атрибута зависит от типа события: это может означать, что событие действительно произошло в этом месте, или может означать, что событие связано с прибытием в это место или отъездом из него

как задать метсоположение

- If the attribute is expressed as an identifier we need a way of interpreting it in order to apply the context. This is done by having a service or database table that can convert between an identifier and a coordinate representation (the conversion from identifier to geographic coordinates is known as geocoding).
- An explicit representation using a coordinate system, for example, a point might be represented as a latitude/longitude pair
- An identifier of a spatial entity, for example, the name of a building or city

several types of spatial context: fixed location, entity distance location, and event distance location.

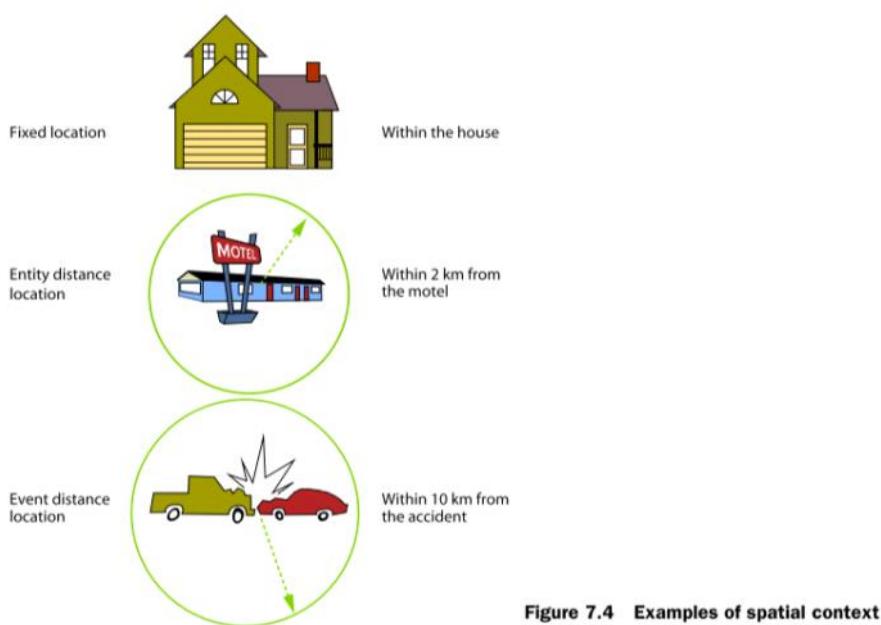


Figure 7.4 Examples of spatial context

(1) Fixed location

- A fixed location context has one or more context partitions, each of which is associated with a spatial entity, sometimes referred to as a **geofence**.
- Событие включается в раздел, если его атрибут местоположения указывает, что оно коррелировано с пространственным объектом раздела
- Экземпляр события классифицируется в контекстном разделе, если его атрибут местоположения каким-либо образом коррелирует с пространственным объектом.
- Гильдия водителей решает расширить приложение Fast Flower Delivery, чтобы узнать, сколько времени водители проводят в определенном районе города. Для этого он использует фиксированный контекст местоположения, определяемый границами этой области, для отслеживания событий GPS-местоположения

водителей . Чтобы определить, какие события произошли в пределах области, контекст использует элемент глобального состояния Neighborhoods .

- Нам все еще нужно объяснить параметр пространственного отношения . Напомним, что местоположение может быть представлено одним из трех типов:
 - точка;
 - линия или
 - ломаная линия; или
 - площадь или
 - объем.

The parameters for the fixed location context are the following:

- *Spatial relation*—The type of comparison used to determine if the event is correlated with the entity
- *Location attribute*—The identifier of the event's location attribute

ally specify one or more explicit partitions. If present, these partition specifications include the following:

- *Partition identifier*—An identifier assigned to this partition. It can be used to refer to this partition from elsewhere.
- *Entity*—Identifier of the spatial entity associated with this partition. This identifier can be used to retrieve the coordinate representation of the entity from the *location service* global state element.

six possible values for spatial relation: contained in, contains, overlaps, disjoint, equals and touches

- При оценке контекста фиксированного местоположения у нас есть два таких местоположения:
 - местоположение события и
 - местоположение объекта.
- Параметр пространственного отношения сообщает нам, как сравнить эти два местоположения, чтобы решить, включать ли экземпляр события в контекстный раздел или нет.

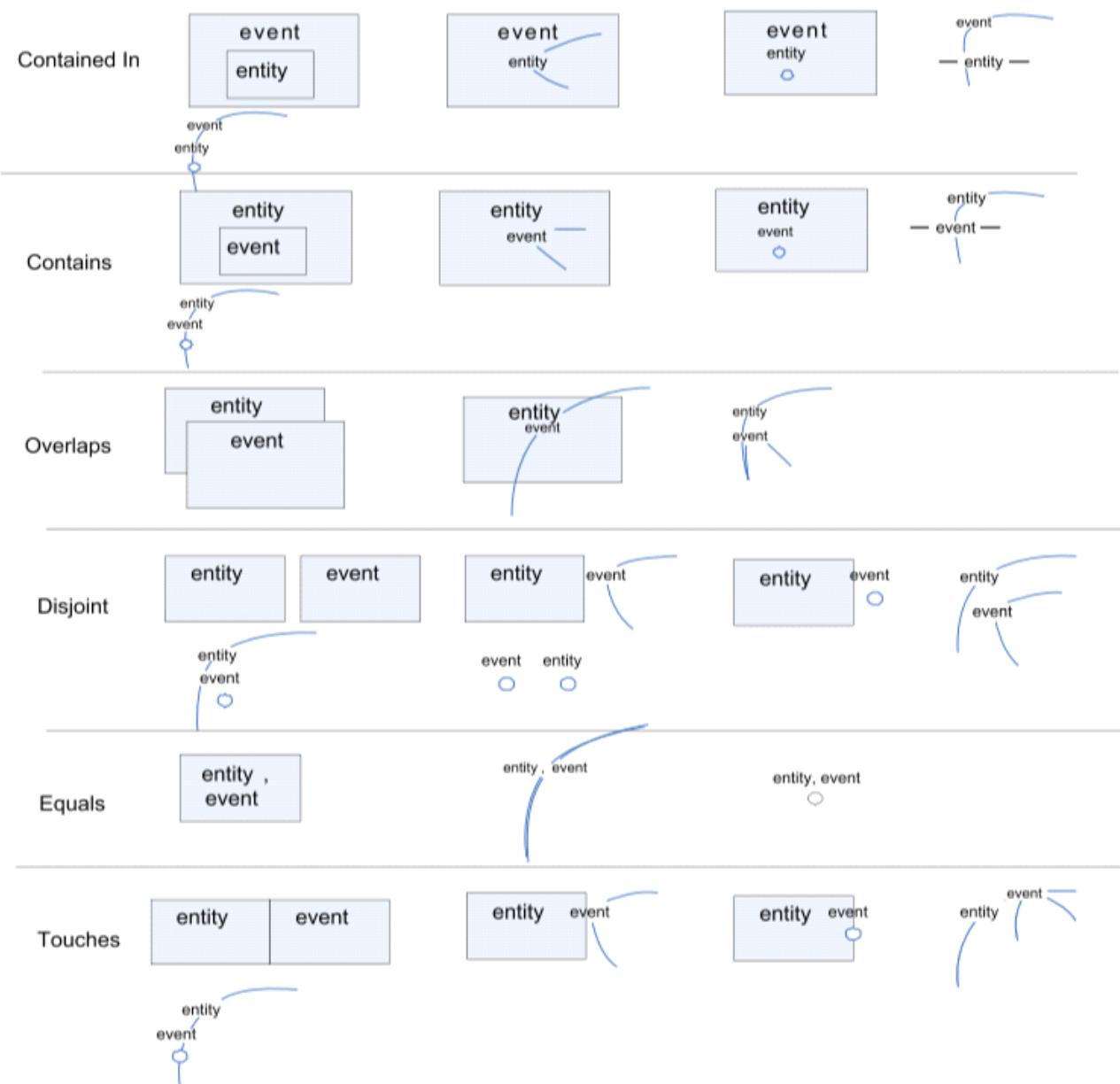


Figure 7.5 Various spatial relation types. Each row illustrates a different relation type, showing examples for which the relation is satisfied.

- The *contains* relation is satisfied if the event location is completely enclosed within the entity location.
- The *overlaps* relation is satisfied if there is an overlap between the entity location and event location, but neither of them is contained within the other.
- The *disjoint* relation is satisfied if there is no overlap between the entity location and event location.
- The *equals* relation is satisfied if the entity location and event location are identical.
- The *touches* relation is satisfied if the event location borders the entity location.

Table 7.1 Combinations of location types that are valid for each spatial relation type

Entity	Contained in	Contains	Overlaps	Disjoint	Equals	Touches
point	line, area			point, line, area	point	line, area
line	line, area	point, line	line, area	point, line, area	line	point, line, area
area	area	point, line, area	line, area	point, line, area	area	point, line, area

(2) Entity distance location

- An entity distance location context gives rise to one or more context partitions, based on the distance between the event's location attribute and some other entity.

This entity may be either stationary or moving.

- o If the entity is moving, the distance relates to the location of the entity at the time that the event occurred (its occurrence time).

Местоположение события и объекта можно указать с помощью типов местоположения точки, линии или области.

- Мы определяем расстояние между двумя местоположениями как кратчайшее расстояние между любой парой точек, взятых из двух местоположений. Это означает, что если вы хотите основывать контекст на расстоянии от центра пространственной области (например, города), вы должны определить местоположение объекта как тип точки
- Пример, показанный на рис. 7.4, где объект - мотель, а контекст используется для отслеживания пожарных тревог в пределах 2 км от мотеля, чтобы предупредить менеджера мотеля о возможной опасности. Этот контекст состоит из одного раздела, и местоположение события и местоположение объекта указаны в виде точек.

The parameters for the *entity distance location* context are as follows:

- *Location attribute*—The identifier of the attribute in the event that gives the event's location
- *Entity attribute*—The identifier of the attribute in the event that gives the identifier of the entity
- *Entity identifier*—The identifier of the entity

more explicit partition specifications. A partition specification includes the following:

- *Partition identifier*—An identifier assigned to this partition. It can be used to refer to this partition from elsewhere.
- *Minimum and maximum distance*—An event is included in the partition if the shortest distance between the event and the entity is less than the maximum distance and not less than the minimum.

(3) Event distance location

- новый раздел создается, когда агент обработки событий получает конкретное событие в качестве входных данных.
- Последующие события любого типа включаются в этот раздел, если они произошли на определенном расстоянии от исходного события.
- Расстояние определяется как в контексте местоположения расстояния до объекта. Данное событие может быть включено более чем в один раздел.

- Обнаружение наличия транспортного средства в радиусе 10 км от места происшествия (список исходных событий - авария , максимальное расстояние - 10 км), как показано на рисунке 7.4.
- События группируются во временном и пространственном контекстах на основе данных самих событий.

The parameters for the *event distance location* context are as follows:

- *Initiator event list*—A new partition is created when the event processing agent receives any of the events specified in the list. An event may be specified by its event type, in which case any instance of that event type will create the partition, or it may be specified by the combination of an event type and a predicate expression. If the predicate is present, the partition will be created only if the event instance also satisfies the predicate (that is to say, the predicate expression must return TRUE when evaluated on the event instance).
- *Location attribute*—The identifier of the attribute in the event that gives the event's location.
- *Minimum and maximum distance*—An event is included in the partition if the shortest distance from the location of the initiator event is less than the maximum distance and not less than the minimum.
- *Context initiator policy*—See section 7.6.

* State-oriented

9 февраля 2021 г. 20:04

- контекст определяется состоянием некоторой сущности, которая является внешней по отношению к системе обработки событий.
- An airport security system could have a threat level status taking values green, blue, yellow, orange, or red. Some events may need to be monitored only when the threat level is orange or above, whereas other events may be processed differently in different threat levels (entity = threat level, relevant states = orange, red).
- контекст контролируется внешним объектом, и решение о том, включать ли событие в раздел, основывается на состоянии внешнего объекта в то время, когда событие происходит или обнаруживается.
- Агент обработки событий, связанный с контекстом, ориентированным на состояние, будет обрабатывать входящие события, только если объект находится в одном из состояний, заданных соответствующим параметром состояний.

The parameters for state-oriented context are as follows:

- *Entity*—The identifier of the external entity whose state controls this context
- *Relevant states*—A list of the external entity states which cause an event to be included in the partition
- *Temporal ordering*—Indicates whether the decision to include or exclude is made using the value of the state at the event instance's occurrence time or at its detection time

* Segmentation-oriented

9 февраля 2021 г. 20:04

- примера рассмотрим агент обработки событий, который принимает один поток входных событий, в котором каждое событие содержит атрибут идентификатора клиента. Значение этого атрибута можно использовать для группировки событий, чтобы для каждого клиента был отдельный контекстный раздел. Каждый контекстный раздел содержит только события, связанные с этим клиентом, поэтому поведение каждого клиента можно отслеживать независимо от других клиентов.
- Here are examples: if the attribute list contains just the driver attribute, events are grouped into partitions according to the value of this attribute, so each driver has its own partition; if the attribute list contains both driver and store, there is a separate partition for each combination of the values of driver and store.
- контекст определяет только атрибут (или атрибуты), который будет использоваться, и это неявно определяет раздел контекста для каждого возможного значения этого атрибута.

text and it has a single parameter:

- *Attribute list*—List of one or more attributes used to determine the context partition

more explicit partitions. If present these partition specifications include the following:

- *Partition identifier*—An identifier assigned to this partition. It can be used to refer to this partition from elsewhere.
- *Partition expressions*—One or more predicate expressions referring to attributes in the event instance. In order to be included in the partition, an event instance must satisfy at least one of these expressions.

As an example of explicit partitions, consider an application where events contain a person's age. The application can use a segmentation context to group together events from different age ranges. To do this it has an explicit partition for each range, for example:

- `age < 21`
- `age ≥ 21 and age < 30`
- `age ≥ 30 and age < 50`
- `age ≥ 50 and age < 67`
- `age ≥ 67`

* Composite contexts

9 февраля 2021 г. 20:05

- Контексты, которые составлены вместе , могут быть любого размера и любого типа, так и на практике они часто бывают разных типов

Набор разделов контекста для составного контекста является декартовым произведением наборов разделов контекстов элементов.

- пример на рисунке Composition of segmentation-oriented context and temporal context
- В примерах предыдущего раздела не имеет значения, в каком порядке мы применяем два контекста

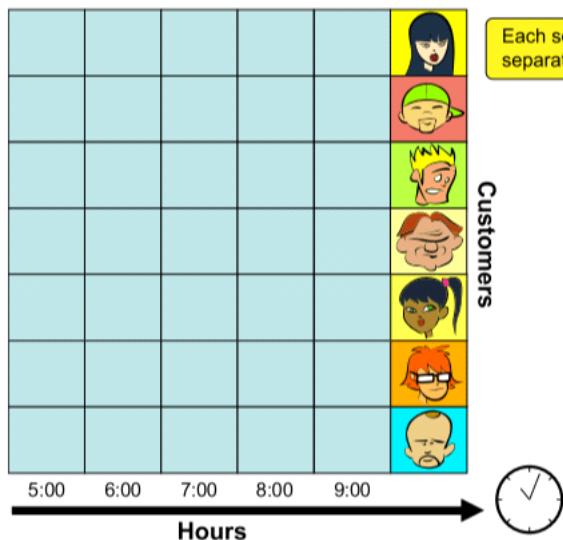


Figure 7.7 An example showing the composition of segmentation-oriented context and temporal context. Each square is a separate context partition and designates the combination of an hour-long interval and a specific customer.

Priority ordering in context composition

- В этом случае порядок влияет на результат,
- Поэтому при определении составного контекста вы можете указать порядок, в котором применяются контексты членов. Мы называем это приоритетным порядком .