! ДОЧИТАТЬ СТАТЬЮ

13 декабря 2020 г. 1:40

https://translate.google.com/translate?hl=&sl=auto&tl=en&u=http%3A%2F%2Fjiagoushi.pro%2Fbook%2Fexport%2Fhtml%2F76
http://jiagoushi.pro/book/export/html/76

и книгу Designing_Event_Driven_Systems_Confluent.pdf

_паттерны

21 декабря 2020 г. 13:57

https://www.confluent.io/blog/building-a-microservices-ecosystem-with-kafka-streams-and-ksql/

local state store

Чтобы проиллюстрировать этот момент, представьте, что у нас есть пользовательский интерфейс, который позволяет пользователям просматривать информацию о заказах, платежах и клиентах в прокручиваемой сетке. Поскольку пользователь может прокручивать отображаемые элементы, время отклика для каждой строки должно быть быстрым.

В традиционной модели без сохранения состояния каждая строка на экране требует вызова всех трех служб. На практике это было бы вялым, поэтому, вероятно, было бы добавлено кеширование, а также некоторый ручной механизм опроса, чтобы поддерживать кеш в актуальном состоянии.

Используя подход потоковой передачи событий, мы можем материализовать данные локально через API Kafka Streams. Мы определяем запрос для данных в нашей сетке: «выберите * из заказов, платежей, клиентов, где…», и Kafka Streams выполняет его, хранит локально, поддерживает актуальность. Это обеспечивает высокую доступность в случае худшего и неожиданного отказа службы

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

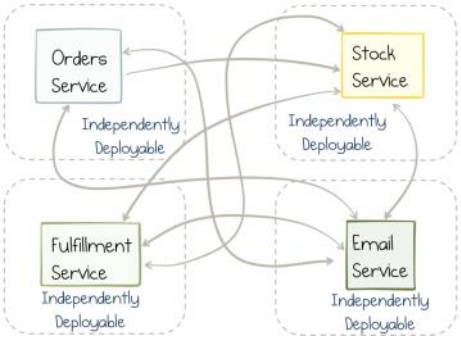
не local state store (?реляционная БД)

Теперь выясняется, что иногда данные необходимо перемещать массово. Иногда сервису требуется локальный исторический набор данных в ядре базы данных по их выбору. Уловка здесь заключается в том, чтобы гарантировать, что копия может быть восстановлена из источника по желанию, путем возврата к распределенному журналу. В этом помогают коннекторы в Kafka.

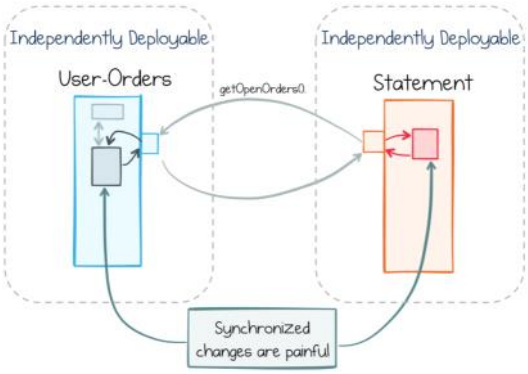
scaling микросервисов в смысле роста команды разработчиков (а не в смысле роста клиентов и нагрузки на микросервисы)

- те один разработчик может поддерживать один а может и 10 микросервисов

Микросервисы можно развертывать независимо. Именно этот атрибут больше, чем какой-либо другой, придает им ценность. Это позволяет им масштабироваться. Расти. Не столько в смысле масштабирования до квадриллионов пользователей или петабайт данных (хотя они могут в этом помочь), сколько в смысле масштабирования с точки зрения людей по мере роста вашей команды и организации.



но проблема координации микросервисных команд между собой остается



в отличие от традиционных БД кафка не использует большие древовидные индексы за которые приходится платить

- журнал имеет O(1) на чтение с концов

- Most traditional message brokers are built using index structures, hash tables or B-trees, used to manage acknowledgements, filter message headers, and remove messages when they have been read. But the downside is that these indexes must be maintained. This comes at a cost. They must be kept in memory to get good performance, limiting retention significantly.
- . Большинство традиционных брокеров сообщений построены с использованием структур индексов, хэш-таблиц или B-деревьев, используемых для управления подтверждениями, фильтрации заголовков сообщений и удаления сообщений после их прочтения. Но обратная сторона - то, что эти индексы необходимо поддерживать. За это приходится платить. Они должны храниться в памяти, чтобы обеспечить хорошую производительность, что значительно ограничивает время хранения. Но журнал имеет значение O (1) при чтении или записи сообщений в раздел, поэтому данные о том, находятся ли данные на диске или кэшированы в памяти, имеют гораздо меньшее значение.

можно масштабировать topic до размера 100 ТБ

Масштабируемость открывает и другие возможности. Отдельные кластеры могут расти до масштабов компании без риска, что рабочие нагрузки перегрузят инфраструктуру. Темы с сотнями терабайт не являются редкостью, а с учетом распределения полосы пропускания между различными службами использование функций мультитенантности упрощает совместное использование кластеров.

хранить все данные в кафке с начала времен это нормально

Идея иметь отдельную копию данных в журнале (особенно если это полная копия) кажется многим расточительной. На самом деле, есть несколько факторов, которые делают это менее важной проблемой. Во-первых, журнал может быть особенно эффективным механизмом хранения. Мы храним более 75 ТБ на каждый центр обработки данных на наших производственных серверах Kafka. Между тем, многим обслуживающим системам требуется гораздо больше памяти для эффективного обслуживания данных (например, текстовый поиск часто полностью выполняется в памяти). Система обслуживания также может использовать оптимизированное оборудование. Например, в большинстве наших систем оперативных данных либо не хватает памяти, либо используются твердотельные накопители. В отличие от этого, система журналов выполняет только линейные операции чтения и записи, поэтому вполне подходит для больших жестких дисков с несколькими ТБ. Наконец, как показано на рисунке выше, в случае, когда данные обслуживаются несколькими системами, стоимость журнала амортизируется по нескольким индексам. Эта комбинация делает расходы на внешний журнал довольно минимальными.

один кластер кафки для всей корпорации это нормально

Архитектуры служб по определению являются мультитенантными. Один кластер будет использоваться множеством разных сервисов. На самом деле нередко все службы в компании используют один кластер. Но это открывает возможность для непреднамеренных атак типа «отказ в обслуживании», вызывающих нестабильность или простои

To help with this, Kafka includes a throughput control feature which allows a defined amount of bandwidth to be allocated to specific services, ensuring that they operate within strictly enforced SLAs.

как еще один из способов повысить надежность сообщений

log.flush.interval.messages = 1 В особо важных случаях использования может потребоваться синхронная загрузка данных на диск

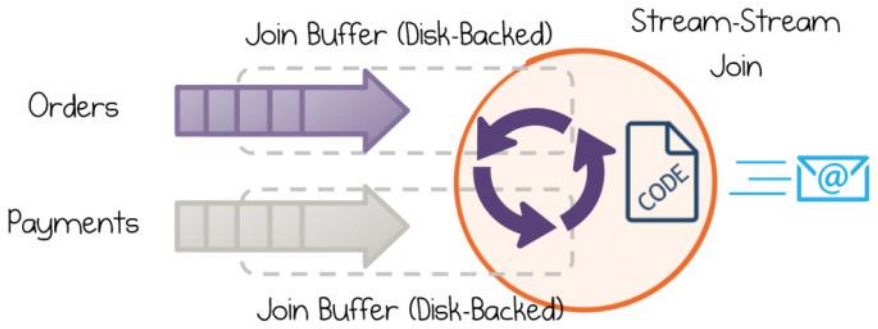
- Highly sensitive use cases may require that data be flushed to disk synchronously. This can be done by setting log.flush.interval.messages = 1 — but this configuration should be used sparingly. It will have a significant impact on throughput, particularly in highly concurrent environments. If you do take this approach, increase the producer batch size, to increase the effectiveness of each disk flush on the broker (batches of messages are flushed together). В особо важных случаях использования может потребоваться синхронная загрузка данных на диск. Это можно сделать, установив log.flush.interval.messages = 1, но эту конфигурацию следует использовать с осторожностью. Это окажет значительное влияние на пропускную способность, особенно в средах с высокой степенью параллелизма. Если вы все же воспользуетесь этим подходом, увеличьте размер пакета производителя, чтобы повысить эффективность каждой очистки диска на посреднике (пакеты сообщений сбрасываются вместе).

_паттерны kafka streams

21 декабря 2020 г. 19:22

join buffer (local state store)

Kafka Streams нуждается в собственном локальном хранилище по нескольким причинам. Наиболее очевидным является буферизация, поскольку в отличие от традиционной базы данных, которая хранит все исторические данные под рукой, потоковым операциям требуется только собирать события за некоторый период времени. Один, который соответствует тому, насколько «поздно» могут относиться друг к другу сообщения.



message enrichment pattern
кафка-таблицы для обогащения даных приходящих из топиков

- Таблицы - это локальное проявление полной темы - обычно сжатой - хранящейся в хранилище состояний по ключу. (Вы также можете думать о них как о потоках с бесконечным сроком хранения .) В контексте микросервисов такие таблицы часто используются для обогащения. Допустим, мы решили включить информацию о Клиенте в нашу логику электронной почты. Мы не можем легко использовать соединение поток-поток, поскольку нет конкретной корреляции между пользователем, создающим заказ, и пользователем, обновляющим свою информацию о клиенте, то есть нет логического верхнего предела того, насколько далеко могут быть друг от друга эти события. Таким образом, для этого стиля работы требуется таблица: весь поток клиентов, начиная со смещения 0, воспроизводится в State Store внутри API Kafka Streams.
- Tables are a local manifestation of a complete topic—usually compacted—held in a state store by key. (You can also think of them as a stream with infinite retention.) In a microservices context, such tables are often used for enrichment. Say we decide to include Customer information in our Email logic. We can't easily use a stream-stream join as there is no specific correlation between a user creating an Order and a user updating their Customer Information—that's to say that there is no logical upper limit on how far apart these events may be. So this style of operation requires a table: the whole stream of Customers, from offset 0, replayed into the State Store inside the Kafka Streams API.

- При использовании KTable хорошо то, что он ведет себя как таблица в базе данных. Поэтому, когда мы присоединяем поток заказов к KTable клиентов, нам не нужно беспокоиться о сроках хранения, окнах или любой другой сложности. Если запись о клиенте существует, соединение будет работать.
- The nice thing about using a KTable is it behaves like a table in a database. So when we join a stream of Orders to a KTable of Customers, there is no need to worry about retention periods, windows or any other such complexity. If the customer record exists, the join will just work.

scatter-gather pattern - собираем три результата событий различных проверок и выпускаем одно событие что заказ-проверен

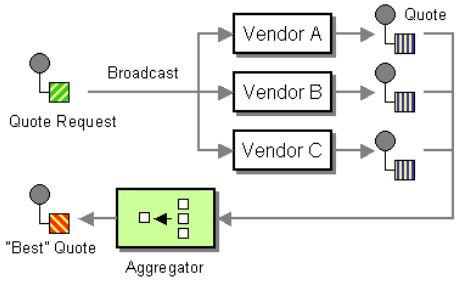
- The result of each validation is pushed through a separate topic, Order Validations, so that we retain the 'single writer' status of the Orders Service —> Orders Topic. The results of the various validation checks are aggregated back in the Order Service (Validation Aggregator) which then moves the 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 рассылает сообщение нескольким получателям и повторно объединяет ответы в одно сообщение.

Scatter-Gather маршрутизирует сообщение с запросом к нескольким получателям. Затем он использует *агрегатор* для сбора ответов и преобразования их в одно ответное сообщение.

EOS (Exactly Once Processing)

21 декабря 2020 г. 19:21

Exactly Once Processing . Это кардинально меняет правила игры для распределенных бизнес-систем, поскольку обеспечивает транзакционные гарантии ХА в масштабируемой форме

? clustered context model

22 декабря 2020 г. 0:35

https://www.confluent.io/blog/build-services-backbone-events/