DB FOR EACH MICROSERVICE

20 декабря 2020 г. 11:33

многочисленные копии одних и тех же центральных данных, чуть ли не в каждом микросервисе

- да действительно данные будут дублироваться в куче БД, но все БД будут иметь общий источник правды в КАФКЕ
  - onenote:///C:\Users\trans\Qsync\vova_from_onenote\tf_algonote_v1\SYSTEMDESIGN\KAFKA__EVENT%20DRIVEN.one#пример%202%20event%20carried%20state%20transfer&section-id={7172B8D4-BDE4-4FFF-A011-59DE79779A94}&page-id={5007A34B-73B1-4EA3-AA07-B0509B4A25D5}&end
  - **все системы могут содержать одни и те же данные** но только в своих разрезах (те подчасть нужную только для конкретной задачи микросервиса)

- **It is safe:** They are all derived from same stream of events
- **Custom projection** just the data each service needs.
- Reduced dependencies
- Low latency

так как во всех микросервисах одни и те же копии данных то может быть проблема несогласованности этих всех многочисленных копий

- Одним из важных последствий передачи 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, в разных микросервисах, которые, если бы они были доступны для записи, могли бы привести к конфликтам и несогласованности.

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

эта БД только кеширует данные
- поэтому эту БД можно назвать local cache

Stateful Stream Processing only caches data, the “golden source” is the shared log, so the problems of data diverging over time are far less prevalen

я должен иметь возможность полностью удалить БД и воссоздать ее заново из событий (входного топика)

- ?собирать в кластер БД для каждого микросервиса?
- особенно в случае ошибки, если что-то пошло не так
- вот то, насколько она быстро поднимется это уже второй вопрос
  - db snapshots
  - ?compacted topics

- One interesting consequence of using event streams as a source of truth is that any data you extract from the log doesn’t need to be stored reliably. If it is lost you can always go back for more. So if the messaging layer remembers, downstream databases don’t have to). This means you can, if you choose, regenerate a database or event-sourced view completely from the log.

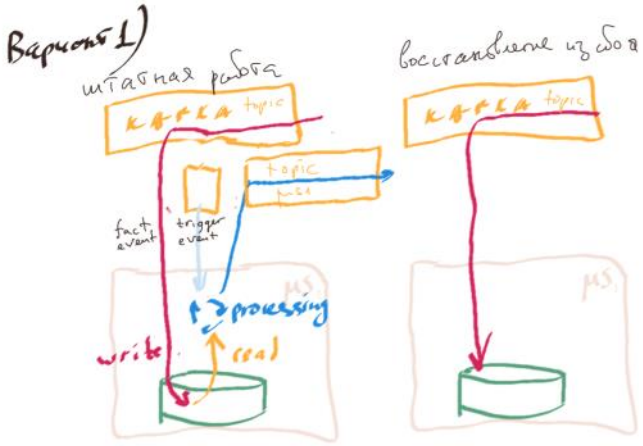
kafka = event sourcing = event store = master системой всех событий
- тогда БД-микросервиса является по-отношению к ней slave
- В традиционной системе источником истины является база данных.

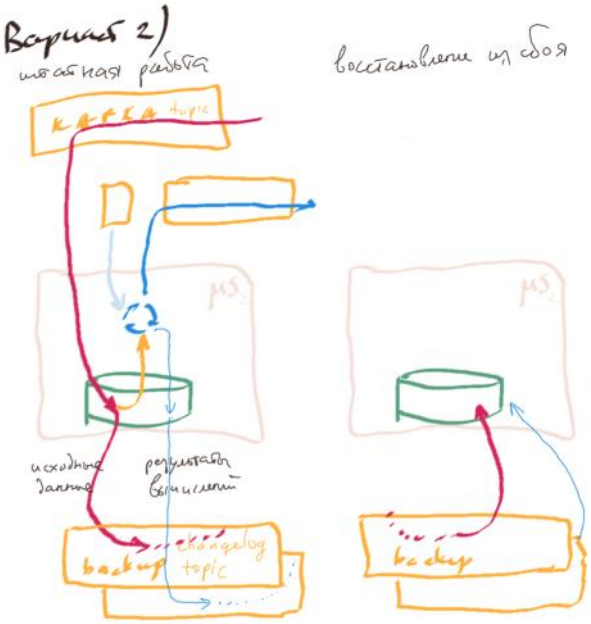
- Общий источник истины оказывается на удивление полезным. Например, микросервисы не делятся своими базами данных друг с другом (это называется антипаттерном IntegrationDatabase). Для этого есть веская причина: базы данных имеют очень богатые API-интерфейсы, которые удивительно полезны сами по себе, 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.”
- Поэтому, когда дело доходит до данных, мы должны недвусмысленно относиться к общим фактам нашей системы. В конце концов, они являются самой сутью нашего бизнеса. Факты могут развиваться со временем, применяться по-разному или даже адаптироваться к разным контекстам, но они всегда должны быть связаны с одной нитью неопровержимой истины, из которой происходят все остальные - центральной нервной системы, которая лежит в основе и движет всеми современный цифровой бизнес.

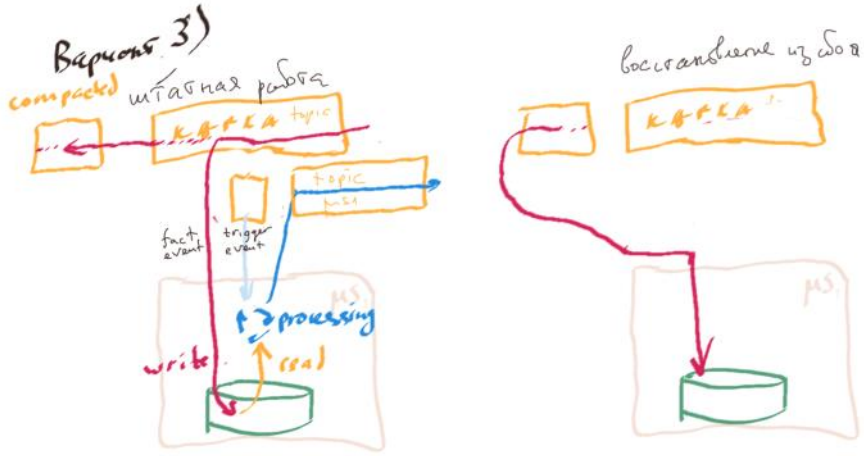
Event Streams as a Shared Source of Truth

каждый микросервис имеет свою БД (те концептуально микросервис stateful)
- но с оговорочкой: centralized stream of events + decentralized stateful stream processing (?? те можно сказать что **стейтфул процессинга** а не самих центральных данных)

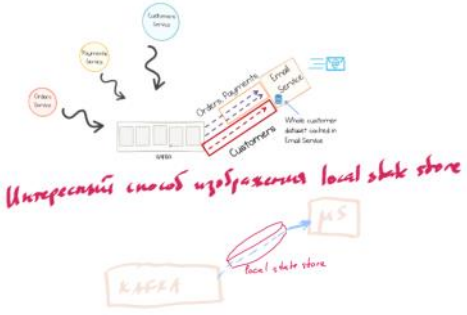
- или можно сделать оговороку: что это **кеш** а не полноценная БД
- или можно сделать оговороку: что это **индекс** (всех данных для скорейшего поиска) а не полноценная БД
- или можно сделать оговороку: что это **вьюха** (но только stateful, materialized view, queryable state, Event-Sourced Views, Derived Views in KStreams State Stores, Interactive Queries) те как бы **query layer**
- те **конкретная вьюха сделана только под конкретную задачу** (но у микросервиса вьюхи может и не быть, если задача не стоит) , что **полностью соотвествует концепции микросервисов (где каждый микросервис тоже сделан под конкретную задачу)**
- или с бекапом обратно в кафку, но если бекап сгорит то должна быть возможность восстановится из исходного топика
- получается что все микросервисы стейтфул (event-driven microservies are stateful)







- .

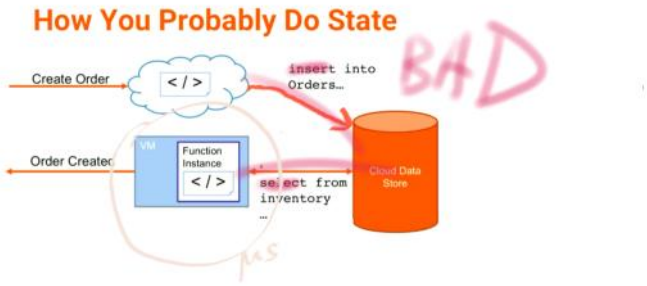


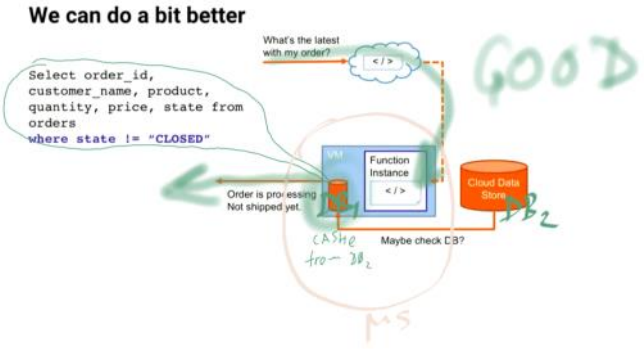
Event-Sourced View - термин для БД микросервиса

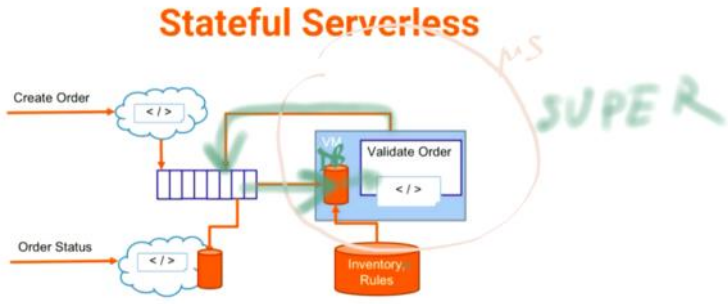
- we will use the term event-sourced view (Figure 7-14) to refer to a query resource (database, memory image, etc.) created in one service from data authored by (and hence owned) by another
- Что отличает представление, основанное на событиях, от типичной базы данных, кеша и т.п., так это то, что, хотя оно может представлять данные в любой форме, которая требуется пользователю, их данные поступают непосредственно из журнала и могут быть восстановлены в любое время.

см пример1 event-carried state transfer: в итоге сервис заказа, всегда держит **все** состояние запасов товаров

- тк микросервис всегда держит актуальное состояние то он по-сути 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?)







(read) БД можно запрашивать локально, те микросервис может читать свою БД по JDBC

- те разрешаются синхронные запросы внутри ограниченного контекста (внутри частей микросервиса)
- но извне, от другого микросервиса не должно быть возможности ПРЯМО запрашивать информацию из БД (другой сервис должен сначала обратится к микросервису а микросервис уже обратится к своей БД)

Поэтому в подходе, основанном на событиях, мы используем события вместо команд. Обработка триггеров событий. Они также превращаются в представления, которые мы можем запрашивать локально. При необходимости мы возвращаемся к удаленным синхронным запросам, особенно в небольших экосистемах, но ограничиваем их область действия в более крупных (в идеале - одним ограниченным контекстом).

(write) писать в БД можно только через ее topic/журнал

- Отличие от традиционной архитектуры баз данных состоит в том, что если вы хотите писать в систему, вы не пишете напрямую в те же базы данных, из которых читаете. Вместо этого вы пишете в журнал, и существует явный процесс преобразования, который берет данные из журнала и применяет их к материализованным представлениям.
- Это разделение чтения и записи - действительно ключевая идея. Помещая записи только в журнал, мы можем сделать их намного проще: нам не нужно обновлять состояние на месте, поэтому мы уходим от проблем одновременной мутации глобального общего состояния. Вместо этого мы просто ведем журнал неизменяемых событий только для добавления. Это дает отличную производительность (добавление к файлу - это последовательный ввод-вывод, который намного быстрее, чем ввод-вывод с произвольным доступом), легко масштабируется (независимые события могут быть помещены в отдельные разделы) и его намного проще сделать надежным.

те запись отделена от чтения

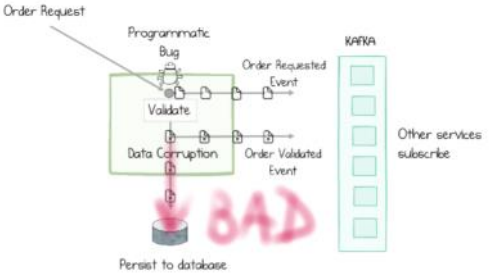
- Нормализованная база данных (без избыточности) оптимизирована для записи, тогда как денормализованная база данных оптимизирована для чтения. Если вы отделите записывающую сторону (журнал) от считывающей (материализованные представления), вы можете денормализовать считывающую сторону в соответствии с вашими пожеланиями, но при этом сохранить способность эффективно обрабатывать записи.

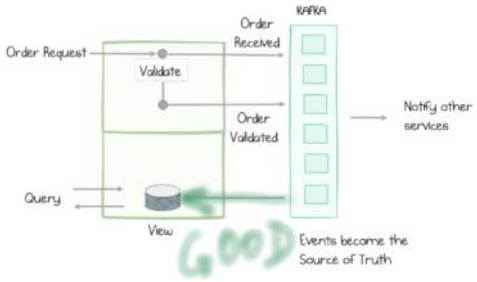
это устраняет все проблемы гонок

- Если материализованные представления обновляются только через журнал, тогда исчезает целый класс условий гонки: журнал определяет порядок, в котором применяются записи, поэтому все представления, основанные на одном журнале, применяют изменения в одном порядке. , по этому в конечном итоге они согласуются друг с другом. Журнал выжимает из потока записи недетерминированность параллелизма.

запись теперь асинхронная (чтение своих собственных данных будет отставать от их записи)

- Загвоздка в том, что, поскольку модель чтения обновляется асинхронно, она будет немного отставать от модели записи во времени. Таким образом, если пользователь выполняет запись, а затем немедленно выполняет чтение, возможно, что запись, которую он написал изначально, не успела распространиться, поэтому они не могут «прочитать свои собственные записи».





confluent

What's Still missing?

- Durable functions everywhere
- Triggers and data from data stores to functions
- Unified view of current state

Events: Approach

Producers:

- **Create a journal of ALL operations on table**
- **Record operation (insert, update, delete)**
- **On creation, queue the journal record to be published**
- **Real time, async, we publish 1 event per journal record**
- **Enable replay by simply requeuing journal record**

Consumers:

- **Store new events in local database, partitioned for fast removal**
- **On event arrival, queue record to be consumed**
- **Process incoming events in micro batches (by default every 250ms)**
- **Record failures locally**

Producers: Database Journal

Document retention period

Code generate journal

Use partitions to manage storage

```
"journal": {
  "interval": "daily",
  "retention": 3
}
```

```
select journal.refresh_journaling('public', 'users', 'journal', 'users');
select partman.create_parent('journal.users', 'journal_timestamp', 'time', 'daily');

update partman.part_config
   set retention = '3 day',
       retention_keep_table = false,
       retention_keep_index = false
 where parent_table = 'journal.users';
```

https://github.com/gilt/db-journaling

несколько таких микро БД у микросервиса **(а не одна большая)**

- каждая БД под свой тип зарпроса

- Сервисам рекомендуется принимать только те данные, которые им нужны, в определенный момент времени (те типы данных впрок не запасать).
- те из ивентов забирать только нужную информацию и отфильтровывать ненужное, чтобы statefult store был маленьким
- те конкретная вьюха сделана только под конкретную задачу

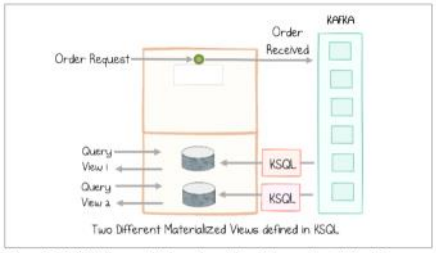


Figure 7-7. CQRS allows multiple read models, which may be optimized for a specific use case, much like a materialized view, or may use a different storage technology

kind of technology should we be using to implement this state in our stream processing application

- *In-process memory*—A mutable variable or variables available inside the stream processing application's own code
- *Local data store*—Some kind of simple data store (for example, LevelDB, RocksDB, or SQLite) that is local to the server or container running this instance of the stream processing application
- *Remote data store*—Some kind of database server or cluster (for example, Cassandra, DynamoDB, or Apache HBase) hosted remotely

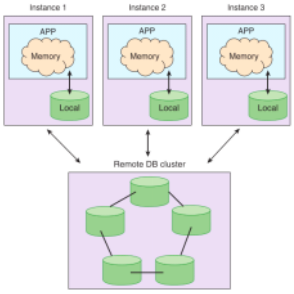


Figure 5.7 A distributed stream-processing application (here with three instances) can keep state in in-process memory, in an instance-local data store, or in a remote data store—or even in a combination of all three.

persistent state (по версии Черняк)

см Persistent State

1) это просто способ обеспечиать отказоустйчивость, в том члучае если кафка лог сгинет навсегда и его повторно нельзя будет обработать (что в случае с кафкой маловероятно)
2) это способ минимимзировать репроцессинг данных из лога, те в противном случае все входные данные пришлось бы обрабатывать повтоно

KAFKA STREAMS 3 МЕХАНИЗМА

1 января 2021 г. 11:20

- Сохранение состояния сопряжено с некоторыми проблемами: при запуске нового узла он должен загрузить все компоненты с отслеживанием состояния (т. Е. Хранилища состояний), прежде чем сможет начать обработку сообщений, а в худшем случае эта перезагрузка может занять некоторое время. Чтобы решить эту проблему, Kafka Streams предоставляет три механизма, которые делают отслеживание состояния более практичным:

Состояние (state) — не что иное, как возможность восстанавливать просмотренную ранее информацию и связывать ее с текущей.

(1) standby replicas (уже несколько ожидающих копий наготове)

- https://kafka.apache.org/10/documentation/streams/developer-guide/config-streams.html#num-standby-replicas
- It uses a technique called standby replicas, which ensure that for every table or state store on one node, there is a replica kept up to date on another. So, if any node fails, it will immediately fail over to its backup node without interrupting processing unduly.
- The number of standby replicas. Standby replicas are shadow copies of local state stores. Kafka Streams attempts to create the specified number of replicas and keep them up to date as long as there are enough instances running. Standby replicas are used to minimize the latency of task failover. A task that was previously running on a failed instance is preferred to restart on an instance that has standby replicas so that the local state store restoration process from its changelog can be minimized. Details about how Kafka Streams makes use of the standby replicas to minimize the cost of resuming tasks on failover can be found in the State section.
- Восстановление представления, основанного на событиях, может оказаться относительно долгим процессом (минуты или даже часы! 2). Из-за этого времени при выпуске нового программного обеспечения разработчики обычно повторно создают представления заранее (или параллельно), причем переключение со старого представления на новое происходит, когда представление полностью занято. По сути, это тот же подход, который используется приложениями обработки потоков с отслеживанием состояния, которые используют потоки Kafka.
- Этот шаблон хорошо работает для простых сервисов, которым нужны наборы данных малого и среднего размера, например, для управления механизмами правил. Работа с большими наборами данных означает более медленное время загрузки. Если вам нужно перестроить наборы данных размером в терабайт в одном экземпляре резидентной базы данных с высокой степенью индексации, это время загрузки может оказаться слишком большим. Но во многих распространенных случаях решения на основе памяти, которые имеют быстрое время записи или горизонтальное масштабирование, сохранят быструю загрузку.

num.standby.replicas - настраивается параметром kafka streams

- onenote:///C:\Users\trans\Qsync\vova_from_onenote\tf_algonote_v1\SYSTEMDESIGN\KAFKA_\KAFKA%20STREAMS.one#distributed%20processing%20and%20fault%20tolerance%20&section-id={422214CB-B11E-4F4F-93A0-EF7DDD1C10A7}&page-id={8C65BBC1-72B7-4C9F-B751-6C65A3AFE2F3}&object-id={26909F81-B939-4A5F-B5E8-1F1AF992DD5D}&12

(2) Disk checkpoints (контрольная точка + загрузить несколько сообщений сверху)

- https://kafka.apache.org/10/documentation/streams/developer-guide/running-app.html#state-restoration-during-workload-rebalance
- Disk checkpoints are created periodically so that, should a node fail and restart, it can load its previous checkpoint, then top up the few messages it missed when it was offline from the log.
- Когда задача переносится, состояние обработки задачи полностью восстанавливается до того, как экземпляр приложения возобновит обработку. Это гарантирует правильные результаты обработки. В Kafka Streams восстановление состояния обычно выполняется путем воспроизведения соответствующей темы журнала изменений для восстановления хранилища состояний. Чтобы минимизировать задержку восстановления на основе журнала изменений с помощью реплицированных локальных хранилищ состояний, вы можете указать num.standby.replicas. Когда потоковая задача инициализируется или повторно инициализируется в экземпляре приложения, ее хранилище состояний восстанавливается следующим образом:
  - Если локального хранилища состояний не существует, журнал изменений воспроизводится с самого раннего до текущего смещения. Это восстанавливает локальное хранилище состояний до самого последнего снимка.
  - Если существует локальное хранилище состояний, журнал изменений воспроизводится с ранее установленного смещения контрольной точки. Изменения применяются, и состояние восстанавливается до самого последнего снимка. Этот метод занимает меньше времени, потому что

он применяет меньшую часть журнала изменений.

(3) compacted topics (более быстрая загрузка из сжатой темы)

- https://kafka.apache.org/documentation.html#compaction
- Finally, compacted topics are used to keep the dataset as small as possible. This acts to reduce the load time for a complete rebuild should one be necessary.

Streams Reset tool (перемещает consumer offsets to "beginning of the topic" для того чтобы заново обработать все входные данн ые)

https://cwiki.apache.org/confluence/display/KAFKA/Kafka+Streams+Application+Reset+Tool
https://www.confluent.io/blog/data-reprocessing-with-kafka-streams-resetting-a-streams-application/

intermediate topics
intermediate topics are user-specified topics that are used as both an input and an output topic within a single application (e.g., a topic that is used i n a call to through()).

internal topics
Internal topics are those topics that are created by the Streams API “under the hood” (e.g., internal repartitioning topics w hich are basically internal intermediate topics).

* Streams Reset tool

1 января 2021 г. 14:22

Streams Reset tool (перемещает consumer offsets to "beginning of the topic" для того чтобы заново обработать все входные данные)

https://cwiki.apache.org/confluence/display/KAFKA/Kafka+Streams+Application+Reset+Tool
https://www.confluent.io/blog/data-reprocessing-with-kafka-streams-resetting-a-streams-application/

- Kafka Streams использует промежуточные темы, которые можно сбросить и повторно создать с помощью инструмента сброса потоков.

kafka-streams-application-reset.sh Используйте этот инструмент **для глобального сброса**, только если приложение полностью остановлено

Как видите, вам нужно только предоставить идентификатор приложения («my -streams-app»), как указано в конфигурации вашего приложения, и имена всех входных и промежуточных тем .
Кроме того, вам может потребоваться указать информацию о подключении Kafka (серверы начальной загрузки) и информацию о подключении Zookeeper.
Как только инструмент сброса приложения завершит свой запуск (и его внутренний потребитель больше не активен), вы можете перезапустить свое приложение как обычно, и теперь оно снова будет обрабатывать свои входные данные с нуля.

```
public class ResetDemo {

    public static void main(final String[] args) throws Exception {
        // Kafka Streams configuration
        final Properties streamsConfiguration = new Properties();
        streamsConfiguration.put(StreamsConfig.APPLICATION_ID_CONFIG, "my-streams-app");
        // make sure to consume the complete topic via "auto.offset.reset = earliest"
        streamsConfiguration.put(ConsumerConfig.AUTO_OFFSET_RESET_CONFIG, "earliest");
        // ...and so on...

        // define the processing topology
        final KStreamBuilder builder = new KStreamBuilder();
        builder.stream("my-input-topic")
            .selectKey(...)
            .through("rekeyed-topic")
            .countByKey("global-count")
            .to("my-output-topic");

         // ...run application...
    }

}
```

```
# After stopping all application instances, reset the application
$ bin/kafka-streams-application-reset --application-id my-streams-app \
                                      --input-topics my-input-topic \
                                       -intermediate-topics rekeyed-topic \
                                      --bootstrap-servers brokerHost:9092 \
                                      --zookeeper zookeeperHost:2181
```

этот ресет не ресетит output topic а только входные и промежуточные темы

It’s important to highlight that, to prevent possible collateral damage, the application reset tool does not reset the output topics of an application. If any output (or intermediate) topics are consumed by downstream applications, it is your responsibility to adjust those downstream applications as appropriate when you reset the upstream application.

можно сделать локальный ресет для одного инстанса а можно сделать сразу для всех инстансов приложения

streams.cleanUp() ; пример **локального сброса** экземпляра через код

```
package io.confluent.examples.streams;

// ...some imports...

public class ResetDemo {

    public static void main(final String[] args) throws Exception {
         // ...prepare your application configuration and processing topology...

        final KafkaStreams streams = new KafkaStreams(builder, streamsConfiguration);

        // Delete the application's local state.
        // Note: In real application you'd call `cleanUp()` only under certain conditions.
        // See Confluent Docs for more details:
        // https://docs.confluent.io/current/streams/developer-guide/app-reset-tool.html#step-2-reset-
        streams.cleanUp();

        streams.start();

        // Add shutdown hook to respond to SIGTERM and gracefully close Kafka Streams
        Runtime.getRuntime().addShutdownHook(new Thread(new Runnable() {
            @Override
            public void run() {
                streams.close();
            }
        }));
```

```
package io.confluent.examples.streams;

// ...some imports...

public class ResetDemo {

    public static void main(final String[] args) throws Exception {
        // ...prepare your application configuration and processing topology...

        final KafkaStreams streams = new KafkaStreams(builder, streamsConfiguration);

        // Delete the application's local state.
        // Note: In real application you'd call `cleanUp()` only under certain conditions.
        // See Confluent Docs for more details:
        // https://docs.confluent.io/current/streams/developer-guide/app-reset-tool.html#step-2-reset-
        streams.cleanUp();

        streams.start();

        // Add shutdown hook to respond to SIGTERM and gracefully close Kafka Streams
        Runtime.getRuntime().addShutdownHook(new Thread(new Runnable() {
            @Override
            public void run() {
                streams.close();
            }
        }));
    }

}
```

вариант сброса, когда проге присваиваем новый application.id (все данные из входного топика будут обработаны с самого начала топика)

- С другой стороны, переименование идентификатора приложения заставляет ваше приложение повторно обрабатывать входные данные с нуля. Почему? Когда используется новый идентификатор приложения, приложение не имеет каких-либо зафиксированных смещений, внутренних тем или локального состояния, связанных с самим собой, потому что все они каким-то образом используют идентификатор приложения для связывания с приложением Kafka Streams. Конечно, вам также необходимо удалить и воссоздать все промежуточные пользовательские темы.
- Во-первых, сброс приложения - это больше, чем просто возможность повторно обработать входные данные. Важной частью сброса является очистка всех внутренних данных.который создается запущенным приложением в фоновом режиме. Например, все внутренние разделы (которые больше не используются) занимают место для хранения в вашем кластере Kafka, если их никто не удаляет. Во-вторых, такая же очистка должна выполняться для данных, записываемых в локальные каталоги состояния.

пример ресета кафка стримс микросервиса

https://www.confluent.io/blog/data-reprocessing-with-kafka-streams-resetting-a-streams-application/

- This application reads data from the **input topic "my-input-topic**", then selects a new record key, and then writes the result into the **intermediate topic "rekeyed-topic"** for the purpose of data re-partitioning. Subsequently, the re-partitioned data is aggregated by a count operator, and the final result is written to the **output topic "my-output-topic"**. Note that in this blog post we don't put the focus on what this topology is actually doing — the point is to have a running example of a typical topology that has input topics, intermediate topics, and output topics.

```
package io.confluent.examples.streams;

// ...some imports...

public class ResetDemo {

    public static void main(final String[] args) throws Exception {
        // Kafka Streams configuration
        final Properties streamsConfiguration = new Properties();
        streamsConfiguration.put(StreamsConfig.APPLICATION_ID_CONFIG, "my-streams-app");
        // make sure to consume the complete topic via "auto.offset.reset = earliest"
        streamsConfiguration.put(ConsumerConfig.AUTO_OFFSET_RESET_CONFIG, "earliest");
        // ...and so on...

        // define the processing topology
        final KStreamBuilder builder = new KStreamBuilder();
        builder.stream("my-input-topic")
            .selectKey(...)
            .through("rekeyed-topic")
            .countByKey("global-count")
            .to("my-output-topic");

        // ...run application...
    }

}
```

creates the **internal topic** my-streams-app-global-count-changelog because the countByKey() operator's name is specified as "global-count".

Рисунок 1: Пример приложения перед его первым запуском (входные смещения равны нулю, пустая промежуточная и выходная тема, без состояния).

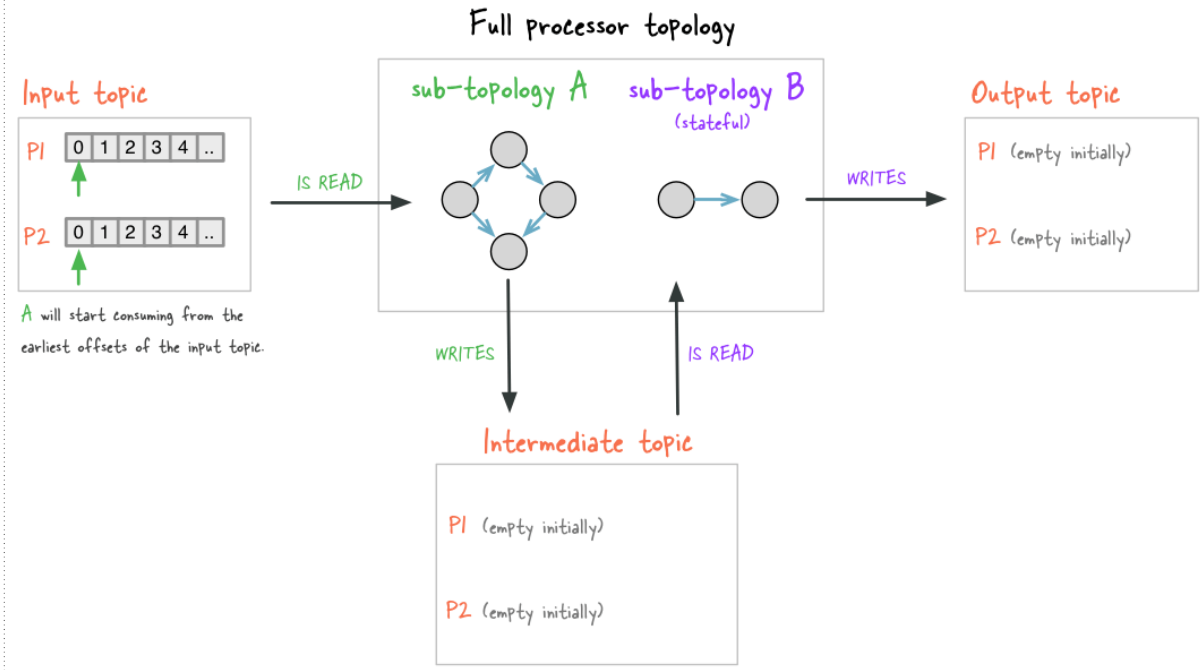
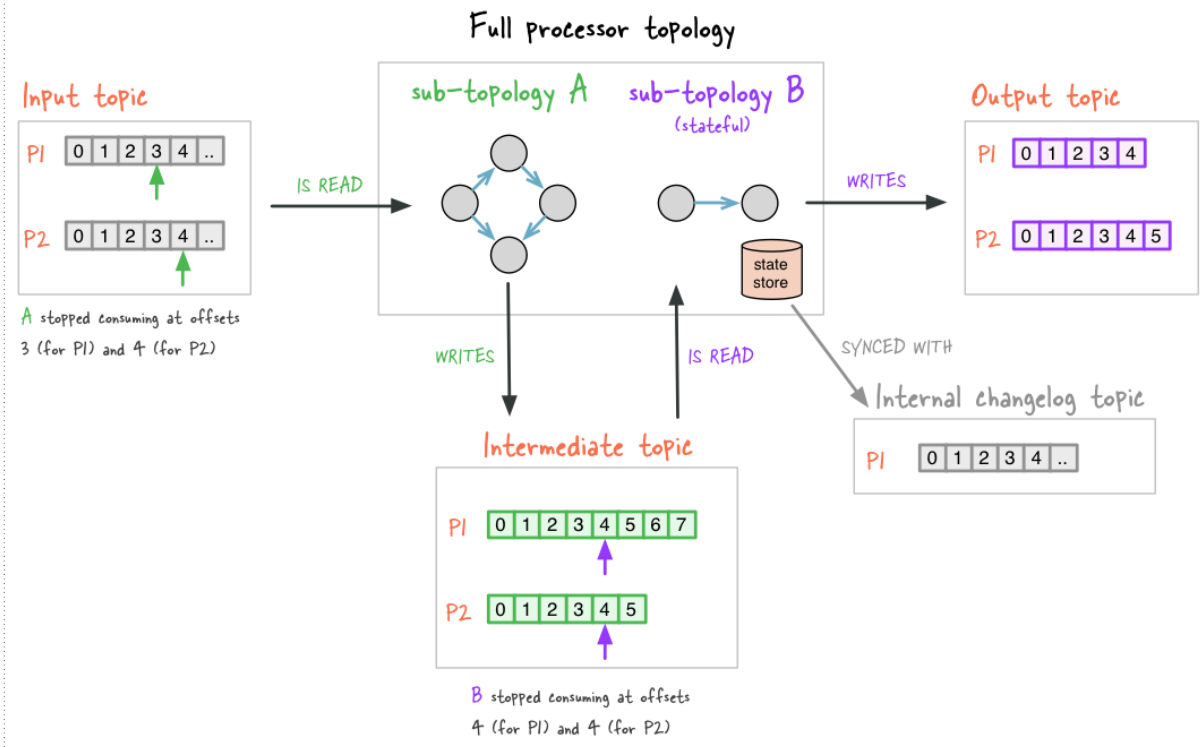


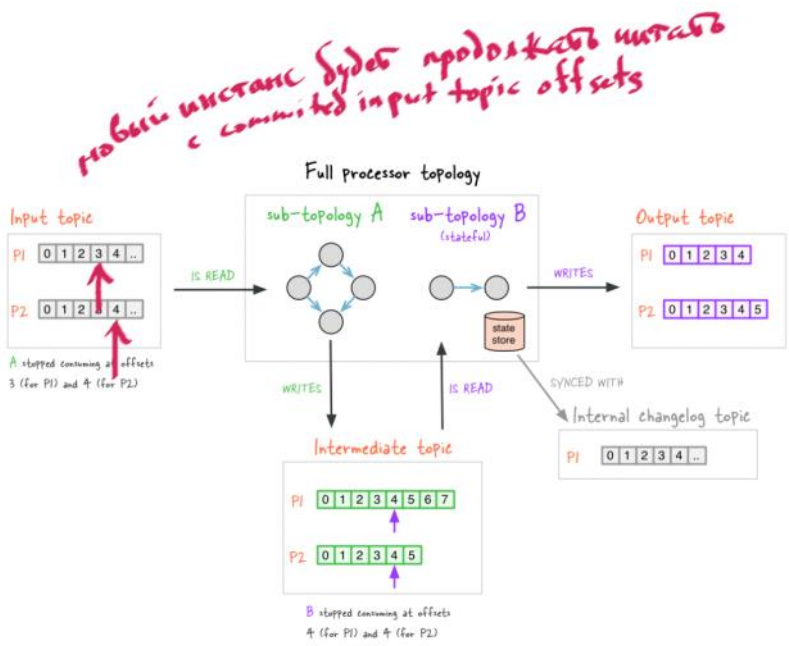
Рисунок 2: Приложение после остановки.

Маленькие вертикальные стрелки обозначают обязательные смещения потребителей для входных и промежуточных тем (цвета обозначаю т соответствующую суб-топологию). Вы можете видеть, например, что субтопология A до сих пор записала в промежуточную тему больше данных, чем субтопология B смогла принять (напри мер, последнее сообщение, записанное в раздел 1, имеет смещение 7, но B только потребила сообщения до смещения 4). Кроме того, субтопология выполняет операции с отслеживанием состо яния и, таким образом, создала локальное хранилище состояний и соответствующий раздел внутреннего журнала изменений для этих хранилищ состояний.



если вдруг инстанс упадет то новый инстанс будет продолжать чистать с того момента на котором остановились (те с закомиченных офсетов входного топика) а НЕ топик с самого начала

Для повторной обработки входных тем с нуля одной из важных проблем является отказоустойчивый механизм Kafka Streams. Если при ложение остановлено и перезапущено, по умолчанию оно не перечитывает ранее обработанные данные снова, но возобновляет обработку с того места, где было остановлено при остановке (см. Зафиксированные смещения темы ввода на рисунке 2). Внутри Streams API использует клиент-клиент Kafka для чтения входных тем и фиксации смещений обработанных сообщений через регулярные промежутки времени (см. Commit.interval.ms). Таким образом, при перезапуске приложение не обрабатывает данные из предыдущего запуска. В случае, если тема уже была полност ью обработана, приложение запустится, но затем будет бездействовать и ждать, пока новые данные будут доступны для обработки.



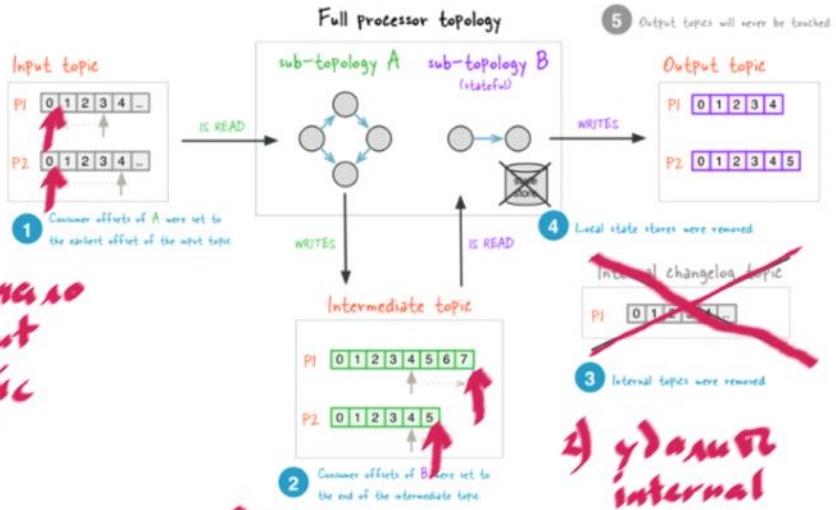
для сброса проги мы дожны переместить офсеты на конец промежуточных тем и "удалить" внутренние changelog топики

- Если имеется несколько субтопологий, может случиться так, что восходящая субтопология создает записи в промежуточные темы быстрее, чем нижележащая субтопология мо жет потреблять (см. Подтвержденные смещения промежуточных тем на рисунке 2). Такая дельта потребления (см. Понятие задержки потребителя в Ka fka) внутри приложения может вызвать проблемы в контексте сброса приложения, потому что после перезапуска приложения нижележащая субтопология возобновит чтение из промежуточных тем с точки, в которой он остановился перед перезапуском. Хотя такое поведение очень желательно при нормальной работе вашего приложения, оно может прив ести к несогласованности данных при сбросе приложения. Поэтому для правильного сброса настроек приложения мы должны указать приложению переходить к самому концу любых промежуточных тем.
  - Промежуточные темы: для промежуточных тем мы должны гарантировать, что не потребляем никаких данных из предыдущих запусков пр иложений. Самый простой и рекомендуемый способ - удалить и воссоздать эти промежуточные темы (напомним, что рекомендуется создавать указанные пользователем темы вручную перед запуском приложения Kafka Streams). Кроме того, необходимо сбросить смещения на ноль для воссозданных тем (так же, как и для тем ввода). Сброс смещений важен, потому что в противном случае приложение подберет эти недопустимые смещения при перезапуске.
  - В качестве альтернативы также можно изменить зафиксированные смещения только для промежуточных тем. Вам следует рассмотреть э тот менее инвазивный подход, когда есть другие потребители для промежуточных тем и, таким образом, удаление тем невозможно. Однако, в отличие от изменения смещений в ходных тем или удаленных промежуточных тем, смещения для сохраненных промежуточных тем должны быть установлены на наибольшее значение (т. Е. На текущий размер журна ла) вместо нуля, таким образом пропуская все еще потребляемые данные. Это гарантирует, что при перезапуске приложение будет использовать только данные из нового запуска (см. № 2 на рисунке 3). Этот альтернативный подход используется инструментом сброса приложения.

- Во-вторых, для любой операции с отслеживанием состояния, такой как агрегирование или объединение, внутреннее состояние этих операций зап исывается в локальное хранилище состояний, которое поддерживается внутренней темой журнала изменений changelog (см. Субтопологию B на рисунке 2). При перезапуске приложения Streams API «обнаруживает» эти темы журнала изменений и любые существующие данные локального состояния, а также гарантирует, что внутреннее состояние полнос тью построено и готово до начала фактической обработки. Поэтому для сброса приложения мы должны также сбросить внутреннее состояние приложения, что означает, что мы должны удалить в се его локальные хранилища состояний и соответствующие им внутренние темы журнала изменений.
  - Внутренние темы: внутренние темы можно просто удалить (см. № 3 на рисунке 3). Поскольку они создаются автоматически Kafka Streams, библиотека может воссоздать их в случае повторной обработки. Как и при удалении промежуточных пользовательских тем, убедитесь, что подтвержденные смещения либо удале ны, либо установлены на ноль.
  - Чтобы удалить эти темы, вам необходимо их идентифицировать. Kafka Streams создает два типа внутренних тем (перераспределение и резервное копирование состояния) и использует следующее соглашение об именах (однако это соглашение об именах может измениться в будущих выпусках, что является одной из причин, по которой мы рекомендуем использовать инструмент сброса приложения, а не вручную сброс ваших приложений):
    <applicationID> - <operatorName> -repartition
    <applicationID> - <operatorName> -changelog
  -

- Подтвержденные смещения входных тем: внутри Kafka Streams использует клиентского клиента Kafka для чтения темы и фиксации смещений обработанных сообщений через регулярные промежутки времени (см. Commit.interval.ms). Таким образом, в качестве первого шага к повторной обработке данных необходимо сбросить зафиксированные смещения. Это можно сделать следующим образом: Напишите специальное клиентское приложение Kafka (например, используя клиентский клиент Java Kafka или любой другой доступный язык), которое использует application.idвашего приложения Kafka Streams в качестве идентификатора группы потребителей. Единственное, что дел ает это специальное клиентское приложение, - это стремиться к нулевому смещению для всех разделов всех входных тем и фиксировать смещение (вы должны отключить автоматическую фиксацию для этого приложения). Поскольку это специальное приложение использует тот же идентификатор группы, что и ваше приложение Kafka Streams (идентификатор приложения используется как идентификатор группы потребителей внутри), фиксация всех смещений на ноль позволяет вашему приложению Streams использовать свои входные темы с нул я при повторном запуске (см. . # 1 на рисунке 3).
- local state store: аналогично внутренним темам, локальные хранилища можно просто удалить (см. № 4 на рисунке 3). Kafka Streams автоматически воссоздает их. Все состояние приложения (экземпляра) хранится в каталоге состояний приложения (см. Параметр state.dir со значением по умолчанию / var / lib / kafka-streams в выпусках Confluent Platform и / tmp / kafka-streams для выпусков Apache Kafka) . В каталоге хранилища состояний каждое приложение имеет собственное дерево каталогов в подпапке, названной по идентификатору приложения. Поэтому самый простой способ удалить хранилище состояний для приложения - это rm -rf <state.dir> / <application.id> (например, rm -rf / var / lib / kafka-streams / my-streams-app).
  - Поскольку сброс локального хранилища состояний встроен в ваш код, для локального сброса не требуется дополнительной работы - локальный сброс включен в перезапуск экземпляра приложения. Для глобального сброса достаточно однократного запуска инструмента сброса приложения.

RESET:



4) output topic не трогаем

3) в начало input topic

4) удалит internal topic

1) в конец intermediate topic

5) удалит DB: local state store

* RDBMS /Elasticsearch

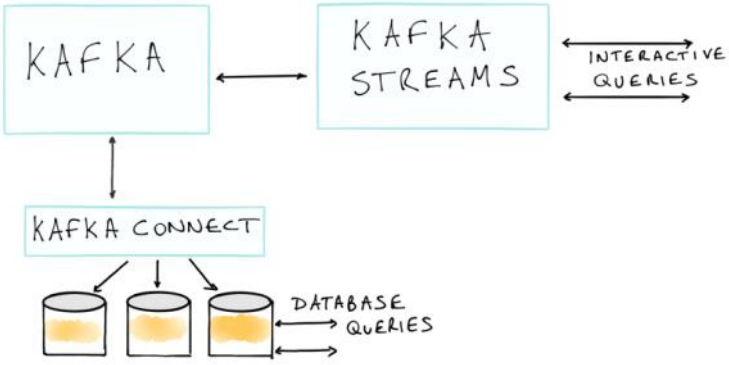
22 декабря 2020 г. 22:02

для чего нужна реляционная БД: local state store нужен только для того чтобы обеспечить гибкость по запросам, которую даёт обычный SQL или Elastic search и не может дать Кафка,.

- подготовка данных и их склейку лучше сделать заранее в кафке а свой денормализованный state store уже просто наполнять и восстанавливать из одного компактед топика 1-в-1
- получается что local state store нужен только для того чтобы обеспечить гибкость по запросам, которую даёт обычный SQL или Elastic search и не может дать Кафка,.

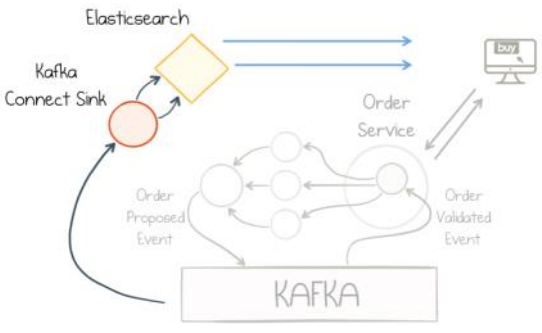
преобразует» физическую базу данных в потоковое приложение благодаря интеграции Kafka Connect с Kafka Streams.

- Kafka Connect будет прослушивать приложение на предмет новых вставок в таблицу (-ы) и помещать соответствующие записи в топик Kafka. Этот же топик будет служить источником для приложения Kafka Streams, так что таблица базы данных превратится в потоковое приложение.
- applications would run their queries against a full-fledged database, which most probably would be continuously updated with the latest processing results through a data flow from your Kafka Streams application to a Kafka output topic, and from there via Kafka Connect to the external database.

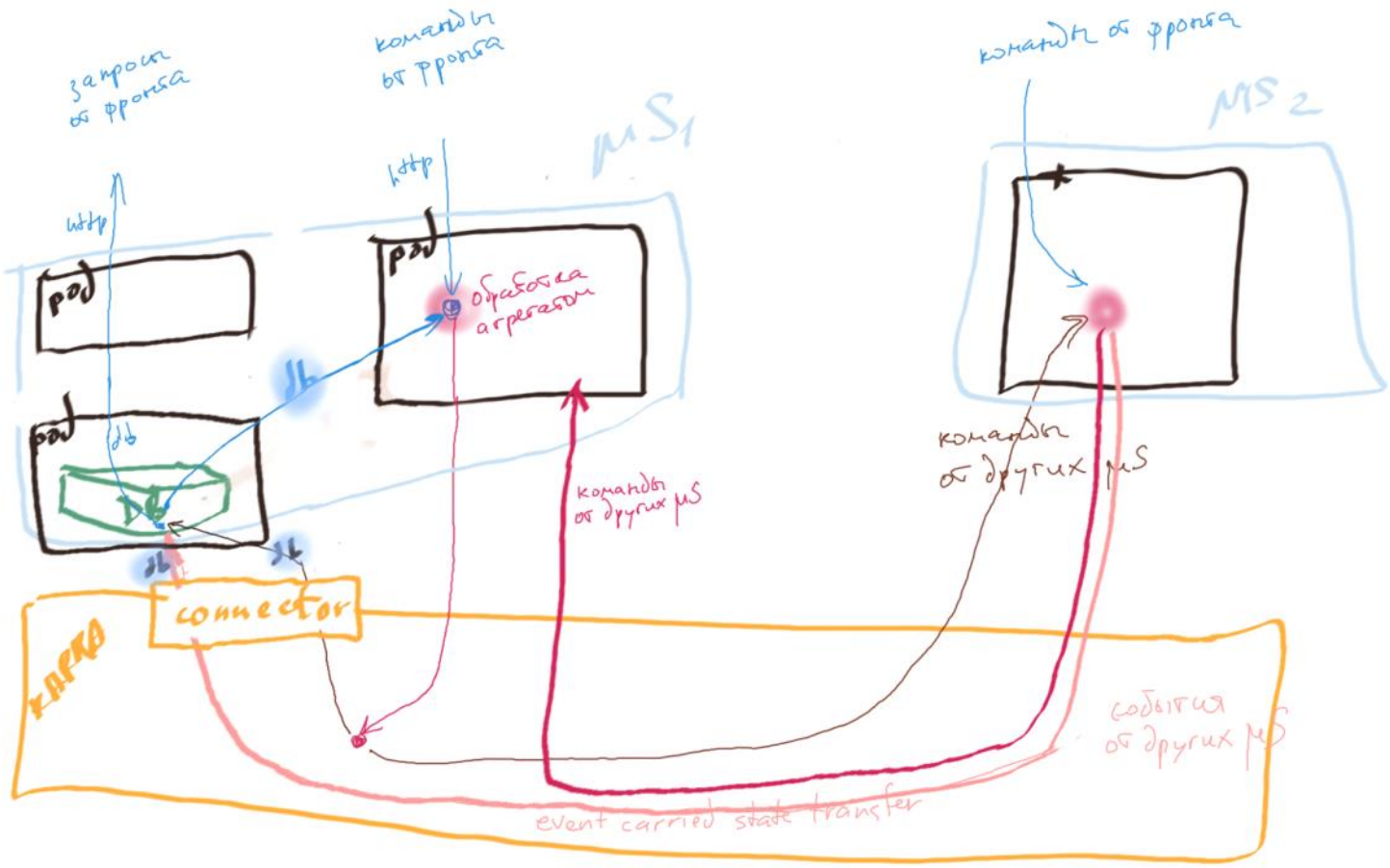


Derived Views created in a Database (как подпаттерн materialized views) в терминологии Confluent

- Kafka поставляется с несколькими коннекторами-приемниками, которые отправляют изменения в самые разные базы данных. Это открывает архитектуру для огромного количества хранилищ данных, настроенных для обработки различных рабочих нагрузок. В приведенном здесь примере мы используем ElasticSearch для его возможностей полнотекстового индексирования, но независимо от формы проблемы, вероятно, существует подходящее хранилище данных.
- Эта способность быстро и эффективно преобразовывать журнал в ряд различных «представлений» оказывается мощным собственным атрибутом. Позже в этой серии мы более внимательно рассмотрим этот шаблон, в частности, различные представления для конкретны х сервисов, которые мы можем создавать.



пример использования kafka sink connector пишущего прямо в local state store микросервиса



Polyglot Views - микросервис может иметь одновременно и RDBMS и NoSQL и ..

- Дополнительным преимуществом использования CQRS является то, что одна модель записи может проталкивать данные во многие модели чтения или материализованные представления. Таким образом, ваша модель чтения может быть в любой базе данных или даже в ряде разных баз данных.
- A replayable log makes it easy to bootstrap such polyglot views from the same data, each tuned to different use cases

Query a Read-Optimized View Created in a Database

- Такие представления можно быстро и легко создать в любом количестве различных баз данных с помощью коннекторов приемников, доступных для Kafka Connect. Как мы обсуждали в предыдущем разделе, их часто называют многоязычными представлениями , и они открывают архитектуру для широкого спектра технологий хранения данных.

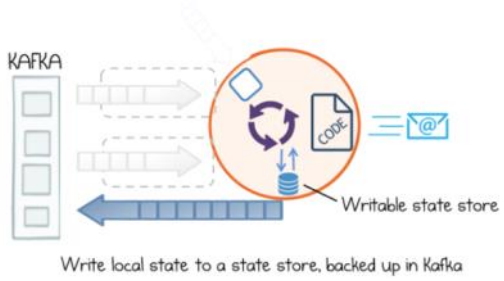
* kafka streams Local RocksDB

22 декабря 2020 г. 17:04

Derived Views in KStreams State Stores (как подпаттерн materialized views) в терминологии Confluent

вариант3) local state store восстанавливается не прямо из основного топика, а из своего бэкап топика (это сделано для скорости)

- те stateful БД позволены но с бекапом в кафку
- или с бекапом обратно в кафку, но если бекап-топик сгорит(разойдется по данным и тп) то должна быть возможность восстановится из исходного топика



(*) 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 kay-value API)
- RocksDB умеет "из-коробки" автоматически себя восстанавливать, в том случае, если вдруг пропадет
- я могу явно задать папку на диске для хранения RoksDB, например persistent volume в кубере

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?

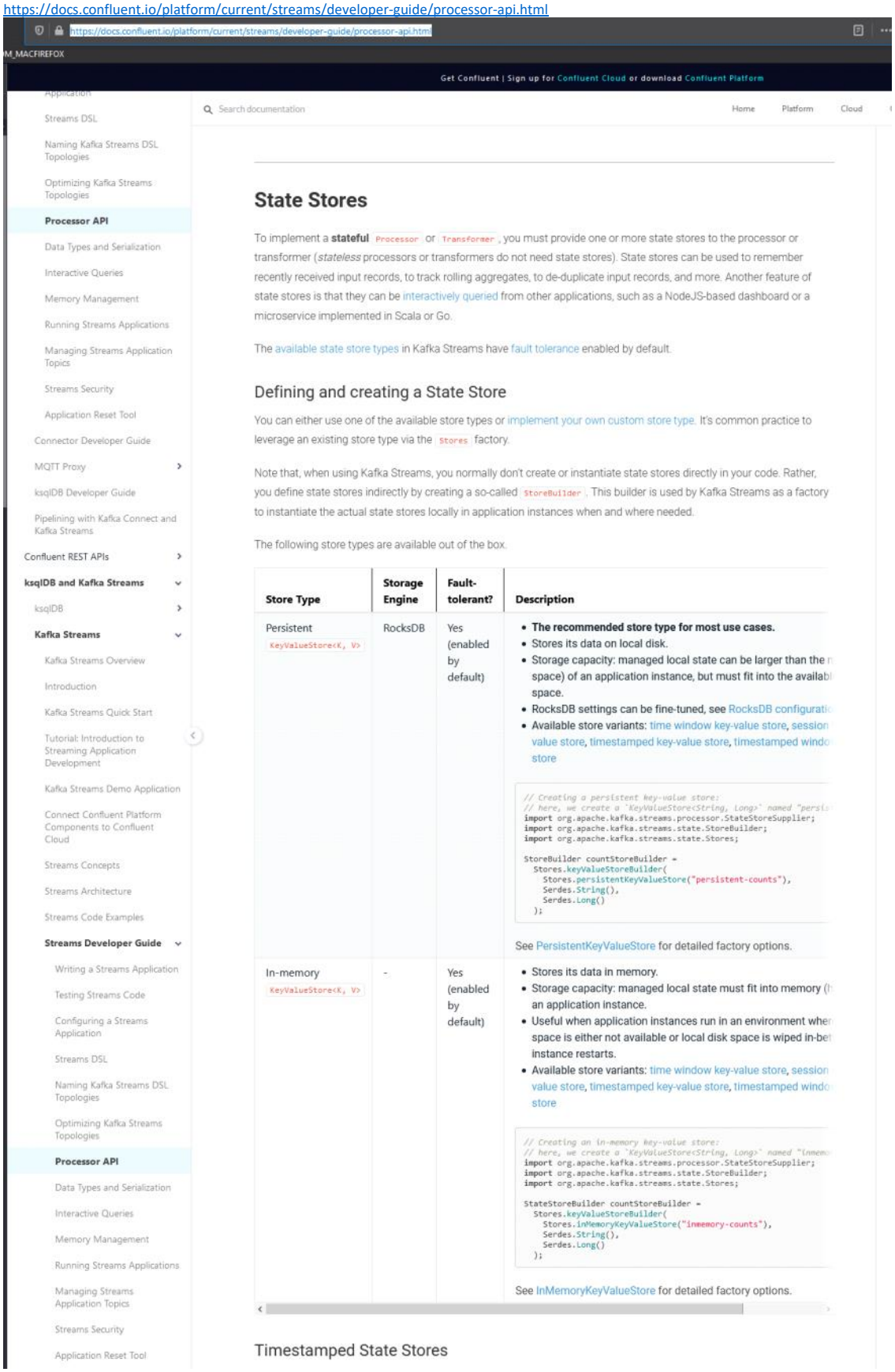
```
private ReadOnlyKeyValueStore<String, Order> ordersStore() {
    return streams.store(
                    ORDERS_STORE_NAME,
                    QueryableStoreTypes.keyValueStore());
}
```

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);
}
```

https://docs.confluent.io/platform/current/streams/developer-guide/processor-api.html



- Note that, when using Kafka Streams, you normally don't create or instantiate state stores directly in your code. Rather, you define state stores indirectly by creating a so-called StoreBuilder. This builder is used by Kafka Streams as a factory to instantiate the actual state stores locally in app lication instances when and where needed.

.withLoggingEnabled() как включить периодический бэкап на local state store (RocksDB) в compacted topic

https://docs.confluent.io/platform/current/streams/developer-guide/processor-api.html#streams-developer-guide-state-store-enable-disable-fault-tolerance

Fault-tolerant State Stores

To make state stores fault-tolerant and to allow for state store migration without data loss, a state store can be continuously backed up to a Kafka topic behind the scenes. For example, to migrate a stateful stream task from one machine to another when elastically adding or removing capacity from your application. This topic is sometimes referred to as the state store's associated *changelog topic*, or its *changelog*. For example, if you experience machine failure, the state store and the application's state can be fully restored from its changelog. You can enable or disable this backup feature for a state store.

By default, persistent key-value stores are fault-tolerant. They are backed by a compacted changelog topic. The purpose of compacting this topic is to prevent the topic from growing indefinitely, to reduce the storage consumed in the associated Kafka cluster, and to minimize recovery time if a state store needs to be restored from its changelog topic.

Similarly, persistent window stores are fault-tolerant. They are backed by a topic that uses both compaction and deletion. Because of the structure of the message keys that are being sent to the changelog topics, this combination of deletion and compaction is required for the changelog topics of window stores. For window stores, the message keys are composite keys that include the "normal" key and window timestamps. For these types of composite keys it would not be sufficient to only enable compaction to prevent a changelog topic from growing out of bounds. With deletion enabled, old windows that have expired will be cleaned up by Kafka's log cleaner as the log segments expire. The default retention setting is `Materialized#withRetention()` + 1 day. You can override this setting by specifying `StreamsConfig.WINDOW_STORE_CHANGE_LOG_ADDITIONAL_RETENTION_MS_CONFIG` in the `StreamsConfig`.

When you open an `Iterator` from a state store you must call `close()` on the iterator when you are done working with it to reclaim resources; or you can use the iterator from within a try-with-resources statement. If you do not close an iterator, you may encounter an OOM error.

Enable or Disable Fault Tolerance of State Stores (Store Changelogs)

You can enable or disable fault tolerance for a state store by enabling or disabling the change logging of the store through `withLoggingEnabled()` and `withLoggingDisabled()`. You can also fine-tune the associated topic's configuration if needed.

Example for disabling fault-tolerance:

```
import org.apache.kafka.streams.state.StoreBuilder;
import org.apache.kafka.streams.state.Stores;

StoreBuilder<KeyValueStore<String, Long>> countStoreSupplier = Stores.keyValueStoreBuilder(
  Stores.persistentKeyValueStore("Counts"),
    Serdes.String(),
    Serdes.Long())
  .withLoggingDisabled(); // disable backing up the store to a changelog topic
```

If the changelog is disabled then the attached state store is no longer fault tolerant and it can't have any standby replicas.

Here is an example for enabling fault tolerance, with additional changelog-topic configuration: You can add any log config from kafka.log.LogConfig. Unrecognized configs will be ignored.

```
import org.apache.kafka.streams.state.StoreBuilder;
import org.apache.kafka.streams.state.Stores;

Map<String, String> changelogConfig = new HashMap<>();
// override min.insync.replicas
changelogConfig.put("min.insync.replicas", "1");

StoreBuilder<KeyValueStore<String, Long>> countStoreSupplier = Stores.keyValueStoreBuilder(
  Stores.persistentKeyValueStore("Counts"),
    Serdes.String(),
    Serdes.Long())
  .withLoggingEnabled(changelogConfig); // enable changelogging, with custom changelog settings
```

local state store микросервиса может содержать какие то свои промежуточные результаты вычислений и мы их можем сохранить в специальный backup-compacted-topic

- это для того чтобы не прогонять события из кафки за ново и не делать вычисления заново
- те при восстановлении микросервиса нам не нужно восстанавливаться из single source of truth
- но все равно данные(уже не события а просто данные вычислений) хранятся не в персистент БД а в кафке

So imagine our service wants to expose JMX metrics that track a variety of operational measures, like the number of orders processed each day. We might keep the running totals for these in memory, then store these values periodically back to Kafka. When the service starts it would load these intermediary values back into memory. As we'll see in the next post, we can use Kafka's Transactions feature to do this in an accurate and repeatable way.

Итак, представьте, что наш сервис хочет предоставить метрики JMX, которые отслеживают различные операционные показатели, такие как количество заказов, обрабатываемых каждый день. Мы могли бы сохранить текущие итоги для них в памяти, а затем периодически сохранять эти значения обратно в Kafka. Когда служба запускается, она загружает эти промежуточные значения обратно в память. Как мы увидим в следующем посте, мы можем использовать функцию транзакций Kafka, чтобы сделать это точным и повторяемым способом.

использовать state store как обычную БД (те для топика мастер системой теперь является не кафка, а БД)

- те если кафка может эмулировать ktable то почему бы ее не использовать заместо обычной таблицы БД

- но мы не должны злоупотреблять: БД не должна получать сообщения извне, источником и single source of truth все равно должна быть кафка и ее события которые попали в микросервис и потом в БД (хоть она и персистится в отдельный топик)

- Конечное использование State Store - сохранение информации, как если бы мы записывали данные в обычную базу данных. Это означает, что мы можем сохранить любую информацию, которую пожелаем, и прочитать ее позже, например, после перезапуска.

вариант использования 1) БД как локальный кеш, а КАФКА как бэкап вычислений для скорости восстановления

- Каждый узел потокового процессора может сохранять собственное состояние
  - Это требуется для буферизации, а также для хранения целых таблиц, например, для обогащения (потоки и таблицы более подробно обсуждаются в разделе « Окна, объединения, таблицы и хранилища состояний» на стр. 135 в главе 14). Идея локального хранилища важна, поскольку она позволяет потоковому процессору выполнять быстрые запросы с сообщениями за раз, не пересекая сеть - необходимая особенность для высокоскоростных рабочих нагрузок, наблюдаемых в сценариях использования в масштабе Интернета. Но эта способность интернализировать состояние в **local stores** ???оказывается полезной и для ряда связанных с бизнесом сценариев использования, как мы обсудим далее в этой книге.

  - например, счетчик-событий требует, чтобы текущая сумма отслеживалась, чтобы в случае сбоя и перезапуска вычисление возобновлялось с предыдущей позиции, а счет оставался точным. Эта способность хранить данные локально концептуально очень похожа на то, как вы могли бы взаимодействовать с базой данных в традиционном приложении. Но в отличие от традиционного двухуровневого приложения, где взаимодействие с базой данных означает выполнение сетевого вызова, при потоковой обработке все состояние является локальным (вы можете рассматривать его как своего рода кеш), поэтому он выполняется быстро. для доступа - никаких сетевых вызовов не требуется. Поскольку он также bacup'ируется обратно в Kafka, он наследует гарантии долговечности Kafka.

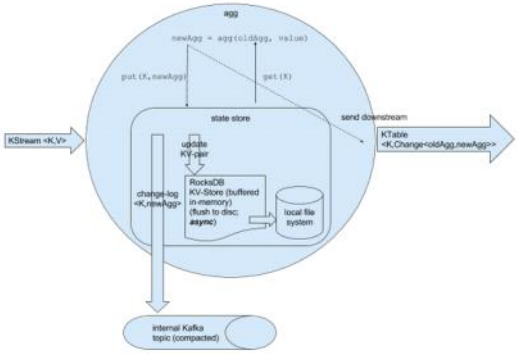
вариант использования 2) бизнес сценарии??

-

```
// Pseudo code
KStream source = builder.stream(...)
source.someTransformation().newKey().someTransformation().someAgg(); // newKey() refers to any transformation that changes the ke
```

Here's an illustration of the above pseudo-code topology:

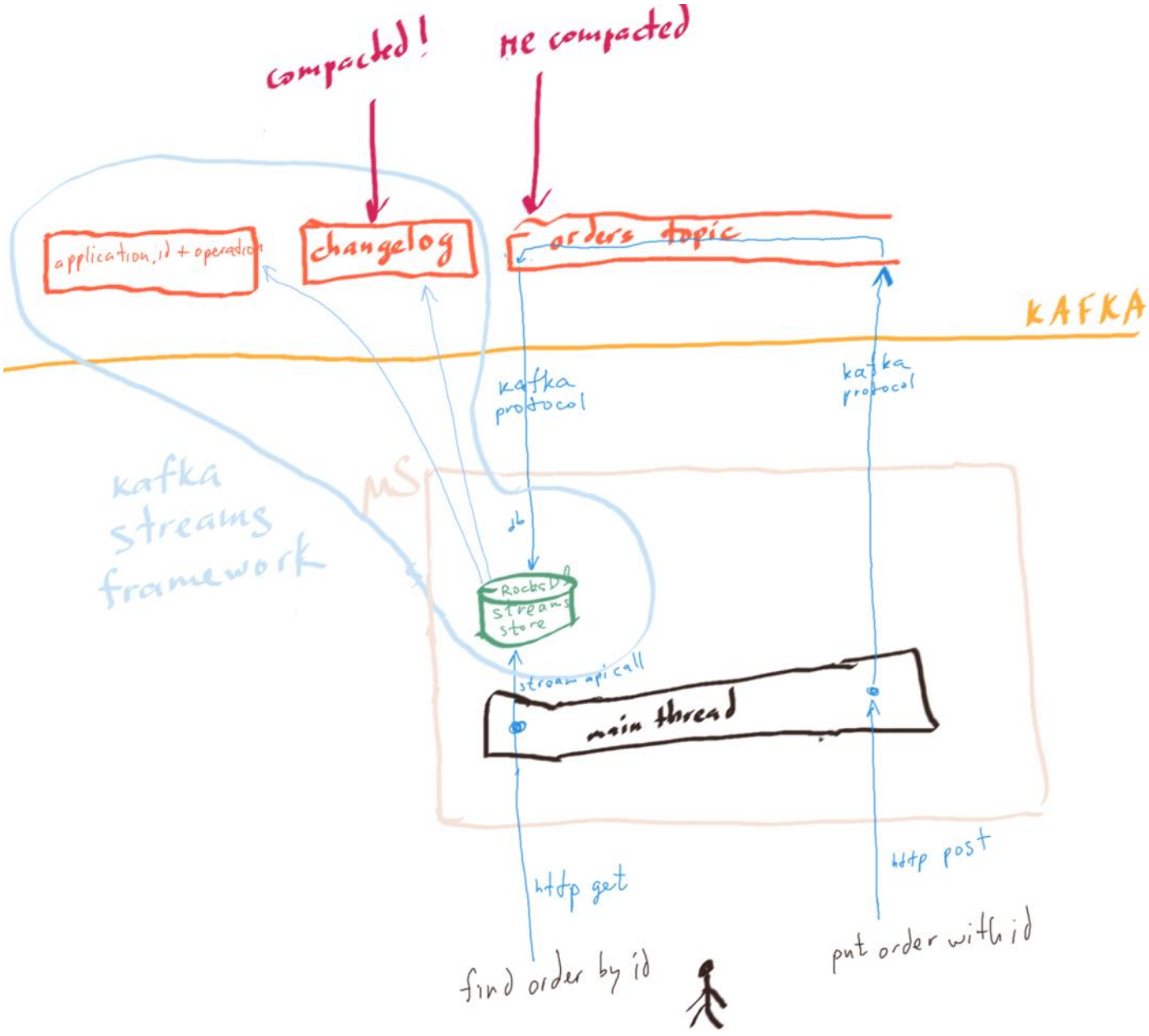
This implies that after "new key" there was no `to()`/`through()` performed. The aggregation itself uses a **RocksDB** instance as key-value **state store** that also persists to local disk. Flushing to disk happens *asynchronously*. Furthermore, an **internal compacted changelog topic** is created. The state store sends changes to the changelog topic in a batch, either when a default batch size has been reached or when the commit interval (see "Commits" below) is reached.



пример 4) используется встроенный в kafka streams: локальный state store RocksDB

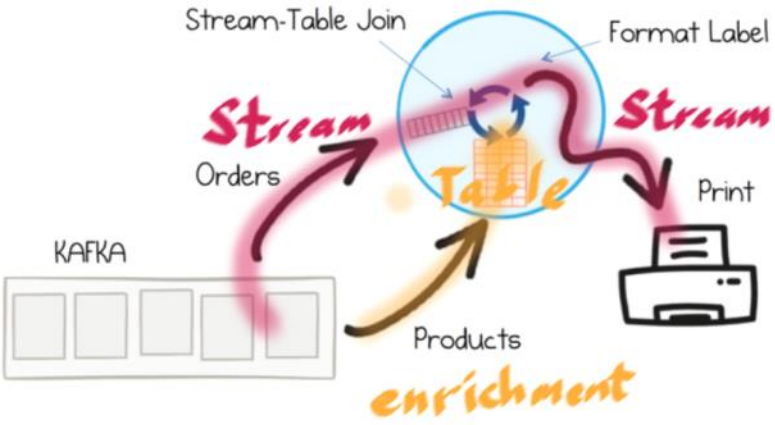
(показаны кишки фреймворка kafka streams)

- ?? rocksdb хранит ту же order topic **но в компактед форме** (и заливает ее для бекапа в changelog топик) **В ЭТОМ РАЗНИЦА МЕЖДУ ORDER TOPIC и BACKUP TOPIC**
- те для быстроты восстанавливаться мы будем не из single source of truth, **но тоже из кафки** из спец бекапа (что тоже хорошо)
- https://cwiki.apache.org/confluence/display/KAFKA/Kafka+Streams+Internal+Data+Management
-



пример 1 результаты джойна НЕ должны быть постоянно доступны, поэтому мы обогощаем события и пробрасываем дальше (стейтфул только временный словарь)

https://www.confluent.io/blog/leveraging-power-database-unbundled/ Example: Building an Embedded, Materialised Views using the Kafka Streams API

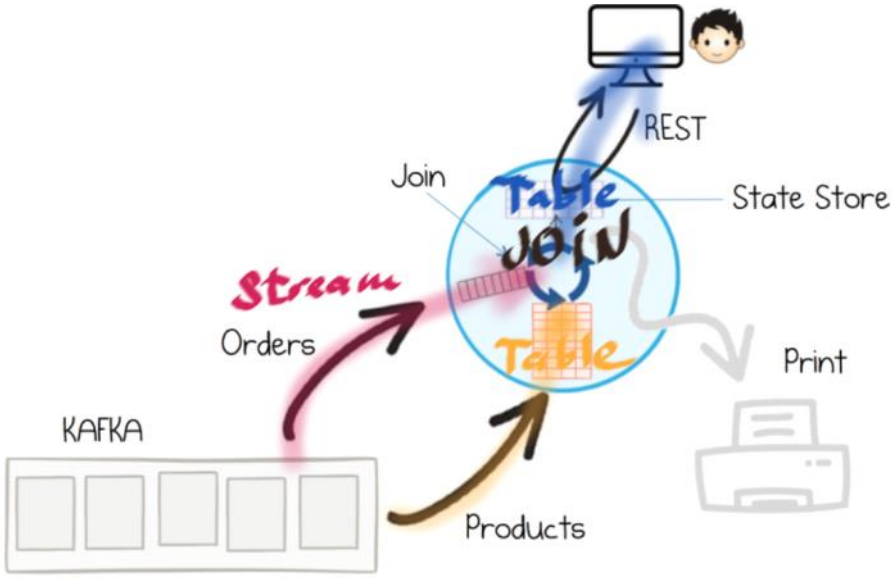



```
//Build a stream from the orders topic
KStream ordersStream = builder.stream(..., “orders-topic”);

//Build a table from the products topic, stored in a local state store called “product-store”
GlobalKTable productsTable = builder.globalTable(...,“product-topic”,“product-store”);

//Join the orders and products to do the enrichment
KStream enrichedOrders = ordersStream.leftJoin(productsTable, (id, order) -> order.productId ...);
```

пример 2 джойним стримы и храним результат в ktable в stateful store




```
//Extending the example above (in the streams thread) push orders into their own topic.
enrichedOrders.through(“enriched-orders-topic”);

//Now build a table from that topic, which will be pushed into the “enriched-orders-store”
KTable enrichedOrdersTable = builder.table(..., “enriched-orders-topic”, “enriched-orders-store”);
```

```
//Inside our request-response thread (e.g. the webserver’s thread)
ReadOnlyKeyValueStore ordersStore = streams.store(“enriched-orders-store”,...);
ordersStore.get(orderId);
```

??? как ..changelog топик шарится между несколькими инстансами микросервиса

- на случай если один инстанс упадет, а другой должен будет подхватить

- Этот подход к управлению состоянием имеет элегантное свойство, состоящее в том, что состояние процессоров также сохраняется в виде журнала. Мы можем думать об этом журнале, как о журнале изменений в таблице базы данных. Фактически, процессоры имеют что-то вроде совместно разделенной таблицы, поддерживаемой вместе с ними. Поскольку это состояние само по себе является журналом, другие процессоры могут подписаться на него. На самом деле это может быть весьма полезно в тех случаях, когда целью обработки является обновление конечного состояния, и это состояние является естественным результатом обработки. This approach to state management has the elegant property that the state of the processors is also maintained as a log. We can think of this log just like we would the log of changes to a database table. In fact, the processors have something very like a co-partitioned table maintained along with them. Since this state is itself a log, other processors can subscribe to it. This can actually be quite useful in cases when the goal of the processing is to update a final state and this state is the natural output of the processing.

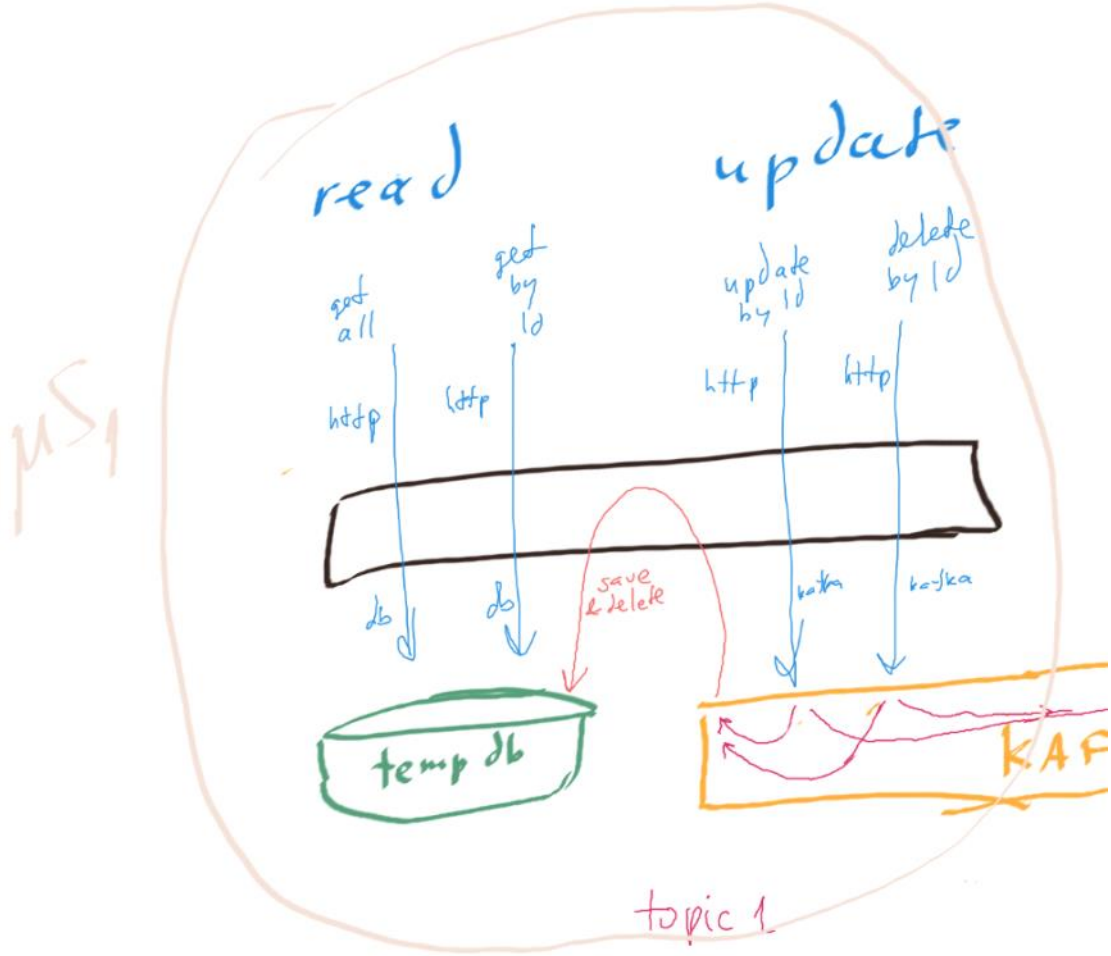
- Одна из более сложных вещей, которую должна делать распределенная система, - это обрабатывать восстановление отказавших узлов или перемещение разделов с узла на узел. Типичный подход заключается в том, что журнал сохраняет только фиксированное окно данных и объединяет его со снимком данных, хранящихся в разделе. В равной степени возможно, чтобы журнал сохранял полную копию данных, а сам журнал собирал мусор . Это перемещает значительную часть сложности с уровня обслуживания, который зависит от системы, в журнал, который может быть общим.

* DB from TOPIC

22 декабря 2020 г. 17:07

пример 1) самописный state store - как отдельная БД

- (см пример (1) new instance with new consumer group id)
- если удалить БД, то она автоматически наполнится из за того что у каждого инстанса новый consumer group id и auto-offset-reset=earliest
- в отличие от итогового варианта 4, эта БД не компактед, потому что топик не компактед (но теоретически можно сделать так чтобы из основного топика с заказами ивенты переливались в спец компактед топик) и из за этого может долго подниматься



нужно выбирать БД оптимизированную для writes (так большое значение имеет скорость восстановления БД с нуля из журнала кафки)

Because worst-case regeneration time is the limiting factor, it helps to pick a write-optimized database or cache. There are a great many options, but sensible choices include:
• An in-memory database/cache like Redis, MemSQL, or Hazelcast
• A memory-optimized database like Couchbase or one that lets you disable journaling like MongoDB
• A write/disk optimized, log-structured database like Cassandra or RocksDB

As a yardstick,
- RocksDB (which Kafka Streams uses) will bulk-load ~10M 500 KB objects per minute (roughly GbE speed).
- Badger will do ~4M × 1K objects a minute per SSD.

- Postgres will bulk-load ~1M rows per minute.

пример с книги криса ричардсона (без восстановления упавшего микросервиса из кафки)

- `OrderHistoryEventHandlers`—Subscribes to events published by the various services and invokes the `OrderHistoryDAO`
- `OrderHistoryQuery API` *module*—Implements the REST endpoints described earlier
- `OrderHistoryDataAccess`—Contains the `OrderHistoryDAO`, which defines the methods that update and query the `ftgo-order-history` DynamoDB table and its helper classes
- `ftgo-order-history` *DynamoDB table*—The table that stores the orders

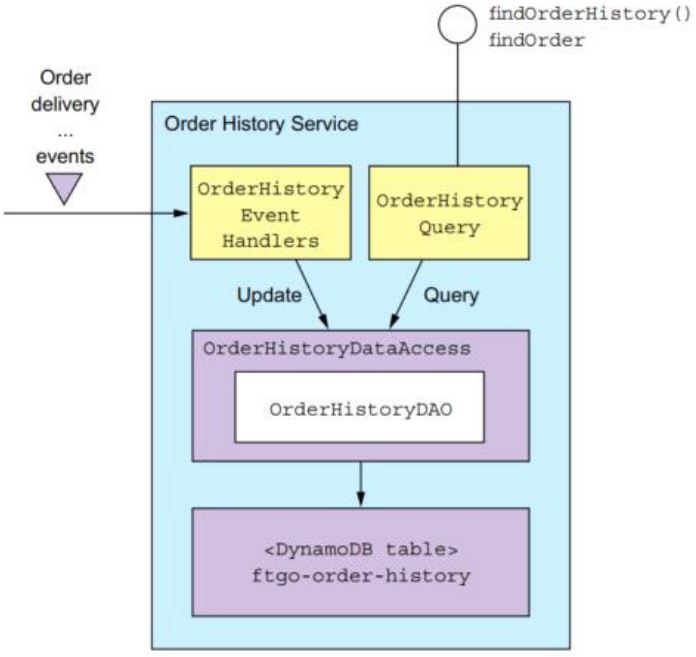


Figure 7.12 The design of `OrderHistoryService`. `OrderHistoryEventHandlers` updates the database in response to events. The `OrderHistoryQuery` module implements the query operations by querying the database. These two modules use the `OrderHistoryDataAccess` module to access the database.

пример ивента обновляющего БД

Listing 7.1 Event handlers that call the `OrderHistoryDao`

```
public class OrderHistoryEventHandlers {

  private OrderHistoryDao orderHistoryDao;

  public OrderHistoryEventHandlers(OrderHistoryDao orderHistoryDao) {
    this.orderHistoryDao = orderHistoryDao;
  }

  public void handleOrderCreated(DomainEventEnvelope<OrderCreated> dee) {
    orderHistoryDao.addOrder(makeOrder(dee.getAggregateId(), dee.getEvent()),
                             makeSourceEvent(dee));
  }

  private Order makeOrder(String orderId, OrderCreatedEvent event) {
    ...
  }

  public void handleDeliveryPickedUp(DomainEventEnvelope<DeliveryPickedUp>
                                       dee) {
   orderHistoryDao.notePickedUp(dee.getEvent().getOrderId(),
           makeSourceEvent(dee));
  }

  ...
```

таблица в микросервисе

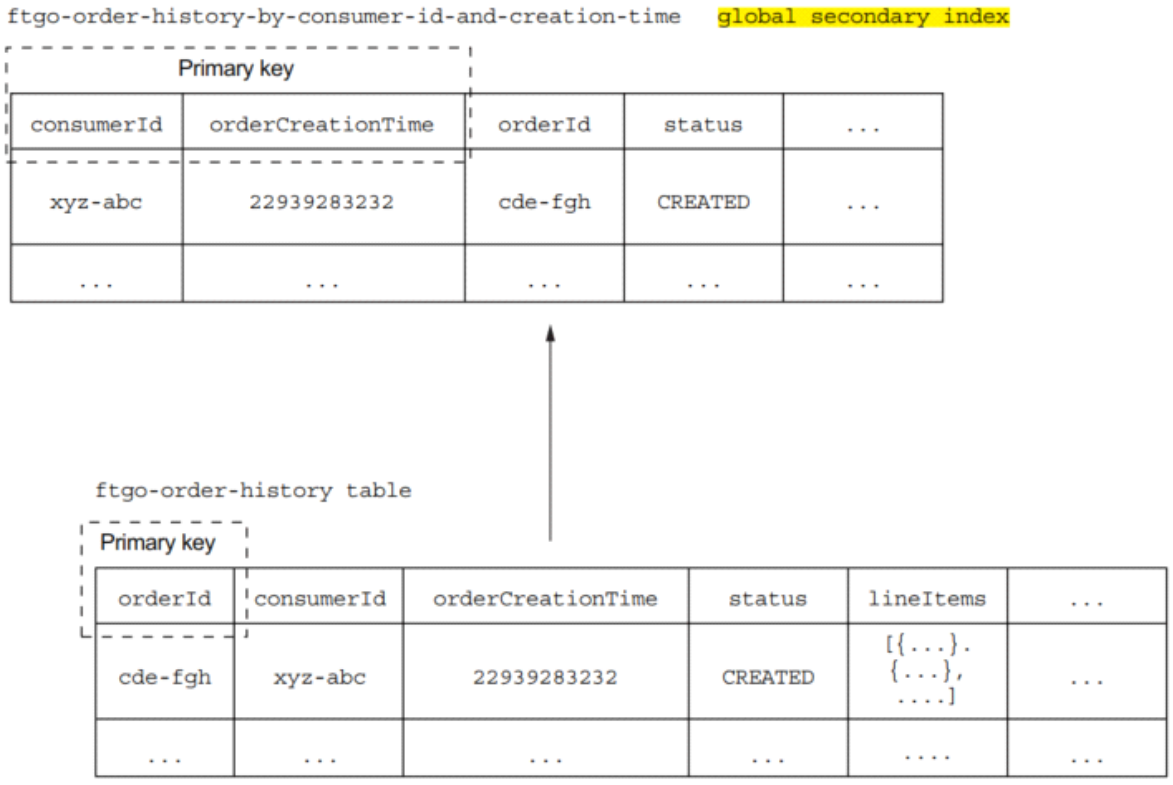


Figure 7.14 The design of the `OrderHistory` table and index

- Было бы логично выбрать в качестве первичного ключа поле orderId. Это позволит сервису Order History вставлять, обновлять и извлекать заказы по их идентификаторам
- Индекс ftgo-order-history-by-consumer-id-and-creation-time позволяет объекту OrderHistoryDaoDynamoDb эффективно извлекать заказы клиента, отсортированные по давности в порядке возрастания
  - Одним из критериев фильтрации является максимальная давность возвращаемых заказов.
- чтобы найти заказы с состоянием CANCELLED, объект OrderHistoryDaoDynamoDb может использовать выражение orderstatus = :orderstatus, где :orderstatus — подставляемый параметр
- Реализация критериев фильтрации по ключевым словам не так проста. Она выбирает заказы, у которых название ресторана или пункты меню совпадают с одним из заданных ключевых слов. Чтобы выполнить поиск по ключевым словам, объект

OrderHistoryDaoDynamoDb разбивает названия ресторанов и пункты меню на термины, которые хранятся в массиве внутри атрибута keywords. Для поиска заказов, соответствующих ключевым словам, он задействует функцию contains(), например contains(keywords, :keywordl) OR contains(keywords, :keyword2), где :keywordl и : keyword2 — подставляемые параметры для заданных ключевых слов

Разбиение результатов запроса на страницы

- У некоторых клиентов будет много заказов. Поэтому логично сделать так, чтобы операция f indOrderHistory () возвращала их постранично. В DynamoDB операции запросов имеют параметр pageSize, который определяет максимальное количество возвращаемых элементов. Если элементов оказывается больше, результат запроса будет содержать ненулевой атрибут LastEvaluatedKey. Чтобы извлечь следующую страницу, объект DAO может указать параметру запроса exclusiveStartKey значение LastEvaluatedKey.
- Как видите, DynamoDB не поддерживает секционное разбиение на страницы. Следовательно, сервис Order History возвращает своему клиенту непрозрачный токен, с помощью которого тот может запросить следующую страницу с результатами.

проблема concurrent/паралельного обновления одного итого же элемента разными событиями (когда они приходят одновременно)

- в DynamoDB нужно использовать upsert/merge (Операция Updateitem) которая автоматически использует блокировки
- Операция Updateitem O обновляет отдельные атрибуты элемента или создает его целиком, если это необходимо. Ее использование имеет смысл, поскольку разные обработчики изменяют разные атрибуты заказа. К тому же эта операция более эффективна, потому что не требует предварительного извлечения заказа из таблицы.

- сервис Order History проблему повторяющихся событий (идемпотентности)
- записывать в элементы события, которые инициировали их обновление и при новом событии сравнивать с записанным

- Для этого объект OrderHistoryDaoDynamoDb может записывать в элементы события, которые инициировали их обновление. Затем он может воспользоваться механизмом условного обновления из операции Updateltem(), чтобы изменять элементы, только если событие не является дубликатом.
- AO-объект OrderHistoryDaoDynamoDb может отслеживать события, полученные из каждого экземпляра агрегата, для этого он использует атрибут "aggregateType”"aggregateId", чье значение равно наивысшему ID принятого события. Если атрибут существует и его значение меньше или равно этому ID событие является дубликатом

```
attribute_not_exists("aggregateType""aggregateId")
     OR "aggregateType""aggregateId" < :eventId
```

Условное выражение позволяет обновить элемент, только если атрибута не существует или `eventId` больше, чем ID последнего обработанного события.

Представьте, к примеру, что обработчик получает из агрегата `Delivery` с ID `3949384394-039434903` событие `DeliveryPickup`, чей идентификатор равен `123323-343434`.

- Отслеживающий атрибут называется `Delivery3949384394-039434903`. Обработчик должен рассматривать событие в качестве дубликата, если значение этого атрибута больше или равно `123323-343434`. Операция `query()`, вызываемая обработчиком событий, обновляет элемент `Order` с помощью условного выражения:

```
attribute_not_exists(Delivery3949384394-039434903)
     OR Delivery3949384394-039434903 < :eventId
```

пример DAO класса

Listing 7.2 The `addOrder()` method adds or updates an `Order`

```
public class OrderHistoryDaoDynamoDb ...

@Override
public boolean addOrder(Order order, Optional<SourceEvent> eventSource) {
 UpdateItemSpec spec = new UpdateItemSpec()
         .withPrimaryKey("orderId", order.getOrderId())
         .withUpdateExpression("SET orderStatus = :orderStatus, " +
                  "creationDate = :cd, consumerId = :consumerId, lineItems =" +
                 " :lineItems, keywords = :keywords, restaurantName = " +
                 ":restaurantName")
```

The primary key of the Order item to update

The update expression that updates the attributes

Listing 7.2 The addOrder() method adds or updates an Order

```
public class OrderHistoryDaoDynamoDb ...

@Override
public boolean addOrder(Order order, Optional<SourceEvent> eventSource) {
 UpdateItemSpec spec = new UpdateItemSpec()
        .withPrimaryKey("orderId", order.getOrderId())
        .withUpdateExpression("SET orderStatus = :orderStatus, " +
                 "creationDate = :cd, consumerId = :consumerId, lineItems =" +
                " :lineItems, keywords = :keywords, restaurantName = " +
                ":restaurantName")
        .withValueMap(new Maps()
                 .add(":orderStatus", order.getStatus().toString())
                .add(":cd", order.getCreationDate().getMillis())
                .add(":consumerId", order.getConsumerId())
                .add(":lineItems", mapLineItems(order.getLineItems()))
                .add(":keywords", mapKeywords(order))
                .add(":restaurantName", order.getRestaurantName())
                .map())
        .withReturnValues(ReturnValue.NONE);
 return idempotentUpdate(spec, eventSource);
}
```

The primary key of the Order item to update

The update expression that updates the attributes

The values of the placeholders in the update expression

Listing 7.3 The notePickedUp() method changes the order status to PICKED_UP

```
public class OrderHistoryDaoDynamoDb ...

@Override
public void notePickedUp(String orderId, Optional<SourceEvent> eventSource) {
 UpdateItemSpec spec = new UpdateItemSpec()
        .withPrimaryKey("orderId", orderId)
        .withUpdateExpression("SET #deliveryStatus = :deliveryStatus")
        .withNameMap(Collections.singletonMap("#deliveryStatus",
               DELIVERY_STATUS_FIELD))
        .withValueMap(Collections.singletonMap(":deliveryStatus",
               DeliveryStatus.PICKED_UP.toString()))
        .withReturnValues(ReturnValue.NONE);
 idempotentUpdate(spec, eventSource);
}
```

Если указать параметр SourceEvent, метод idempotentUpdate() сделает вызов SourceEvent .addDuplicateDetection(), чтобы добавить в UpdateItemSpec условное выражение, описанное ранее. Метод idempotentllpdate() перехватывает и игнорирует исключение ConditionalCheckFailedException, которое генерируется вызовом updateItem() в случае, если событие является дубликатом

Listing 7.4 The idempotentUpdate() method ignores duplicate events

```
public class OrderHistoryDaoDynamoDb ...

private boolean idempotentUpdate(UpdateItemSpec spec, Optional<SourceEvent>
        eventSource) {
 try {
  table.updateItem(eventSource.map(es -> es.addDuplicateDetection(spec))
          .orElse(spec));
  return true;
 } catch (ConditionalCheckFailedException e) {
  // Do nothing
  return false;
 }
}
```

пример запроса из БД

Listing 7.5 The findOrderHistory() method retrieves a consumer's matching orders

```
public class OrderHistoryDaoDynamoDb ...

@Override
public OrderHistory findOrderHistory(String consumerId, OrderHistoryFilter
        filter) {

 QuerySpec spec = new QuerySpec()
        .withScanIndexForward(false)
        .withHashKey("consumerId", consumerId)
        .withRangeKeyCondition(new RangeKeyCondition("creationDate")
                .gt(filter.getSince().getMillis()));

 filter.getStartKeyToken().ifPresent(token ->
       spec.withExclusiveStartKey(toStartingPrimaryKey(token)));

 Map<String, Object> valuesMap = new HashMap<>();
```

Specifies that query must return the orders in order of increasing age

The maximum age of the orders to return

Listing 7.5 The `findOrderHistory()` method retrieves a consumer's matching orders

```
public class OrderHistoryDaoDynamoDb ...

@Override
public OrderHistory findOrderHistory(String consumerId, OrderHistoryFilter
        filter) {

 QuerySpec spec = new QuerySpec()
         .withScanIndexForward(false)
          .withHashKey("consumerId", consumerId)
         .withRangeKeyCondition(new RangeKeyCondition("creationDate")
                                .gt(filter.getSince().getMillis()));

 filter.getStartKeyToken().ifPresent(token ->
       spec.withExclusiveStartKey(toStartingPrimaryKey(token)));

 Map<String, Object> valuesMap = new HashMap<>();

 String filterExpression = Expressions.and(
          keywordFilterExpression(valuesMap, filter.getKeywords()),
         statusFilterExpression(valuesMap, filter.getStatus()));

 if (!valuesMap.isEmpty())
  spec.withValueMap(valuesMap);

 if (StringUtils.isNotBlank(filterExpression)) {
  spec.withFilterExpression(filterExpression);
 }

 filter.getPageSize().ifPresent(spec::withMaxResultSize);

 ItemCollection<QueryOutcome> result = index.query(spec);

 return new OrderHistory(
         StreamSupport.stream(result.spliterator(), false)

            .map(this::toOrder)
             .collect(toList()),
         Optional.ofNullable(result
               .getLastLowLevelResult()
               .getQueryResult().getLastEvaluatedKey())
            .map(this::toStartKeyToken));
}
```

Specifies that query must return the orders in order of increasing age

The maximum age of the orders to return

Construct a filter expression and placeholder value map from the OrderHistoryFilter.

Limit the number of results if the caller has specified a page size.

Create an Order from an item returned by the query.

* inmemDB from TOPIC (memory image)

22 декабря 2020 г. 21:56

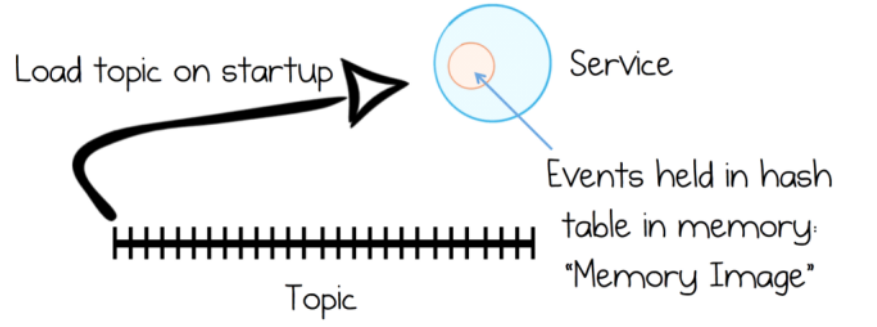
паттерн аналогичен обычному local state store, только БД в памяти

In Memory Views (как подпаттерн materialized views) в терминологии Confluent

memory image паттерн Мартина фаулера
Prepopulated Caches паттерн

подразумевается что ВЕСЬ local state store находится в памяти микросервиса

- для кэширования всего набора данных в память, где его можно запросить, вместо использования внешней базы данных.
- Источники событий часто используются с образами памяти (memory image) . Это просто причудливый термин, придуманный Мартином Фаулером, для кэширования всего набора данных в память, где его можно запрашивать, вместо использования внешней базы данных.
- Образы памяти предоставляют простую и эффективную модель для наборов данных, которые (а) помещаются в память и (б) могут быть загружены в разумные сроки. Например, вышеприведенная служба проверки может загрузить в память один миллион продуктов, скажем, по 100 Б каждый. Это займет около 100 МБ ОЗУ и загрузится из Kafka примерно за секунду на GbE. В наши дни память, как правило, легче масштабировать, чем пропускная способность сети, поэтому «время загрузки наихудшего случая» часто является более ограничивающим фактором из двух.



для восстановления БД из топика лучше использовать cpmpacted topic

Чтобы уменьшить время загрузки, полезно сохранять моментальный снимок журнала событий, используя сжатую тему (о которой мы говорили в предыдущем посте). Это представляет собой «последний» набор событий без какой-либо «истории версий».

этот паттерн для высокоскоростных систем но с небольшим объемом данных

Образы памяти обеспечивают оптимальное сочетание быстрого чтения в процессе с производительностью записи, которая намного выше и более масштабируема, чем у типичной базы данных. Таким образом, он особенно хорошо подходит для высокопроизводительных сценариев использования.

kafka streams - преобразование стрима событий из топика в ktable (самый простой способ сделать таблицу микросервиса)

- В простейшем случае это включает в себя превращение потока событий в Kafka в таблицу, которую можно запрашивать локально. Например, для преобразования потока событий Customer в таблицу Customer, которую можно запросить с помощью CustomerId, требуется всего одна строка кода

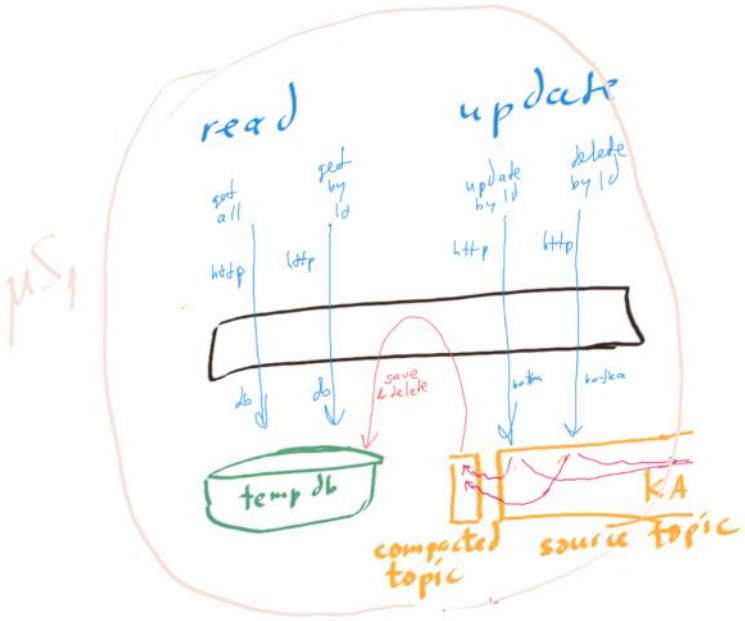
```
KTable<CustomerId, Customer> customerTable = builder.table("customer-topic");
```

* DB from COMPACTED topic

22 декабря 2020 г. 17:08

для чего нужна реляционная БД: local state store нужен только для того чтобы обеспечить гибкость по запросам, которую даёт обычный SQL или Elastic search и не может дать Кафка,.

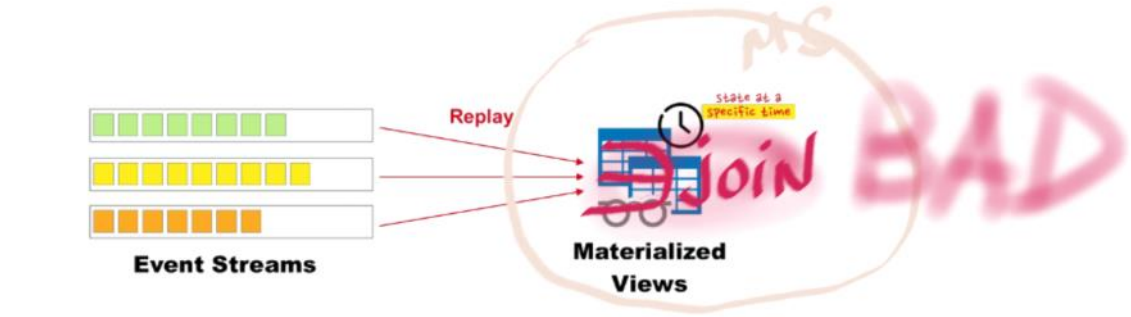
- подготовка данных и их склейку лучше сделать заранее в кафке а свой денормализованный state store уже просто наполнять и восстанавливать из одного компактед топика 1-в-1
- получается что local state store нужен только для того чтобы обеспечить гибкость по запросам, которую даёт обычный SQL или Elastic search и не может дать Кафка,.

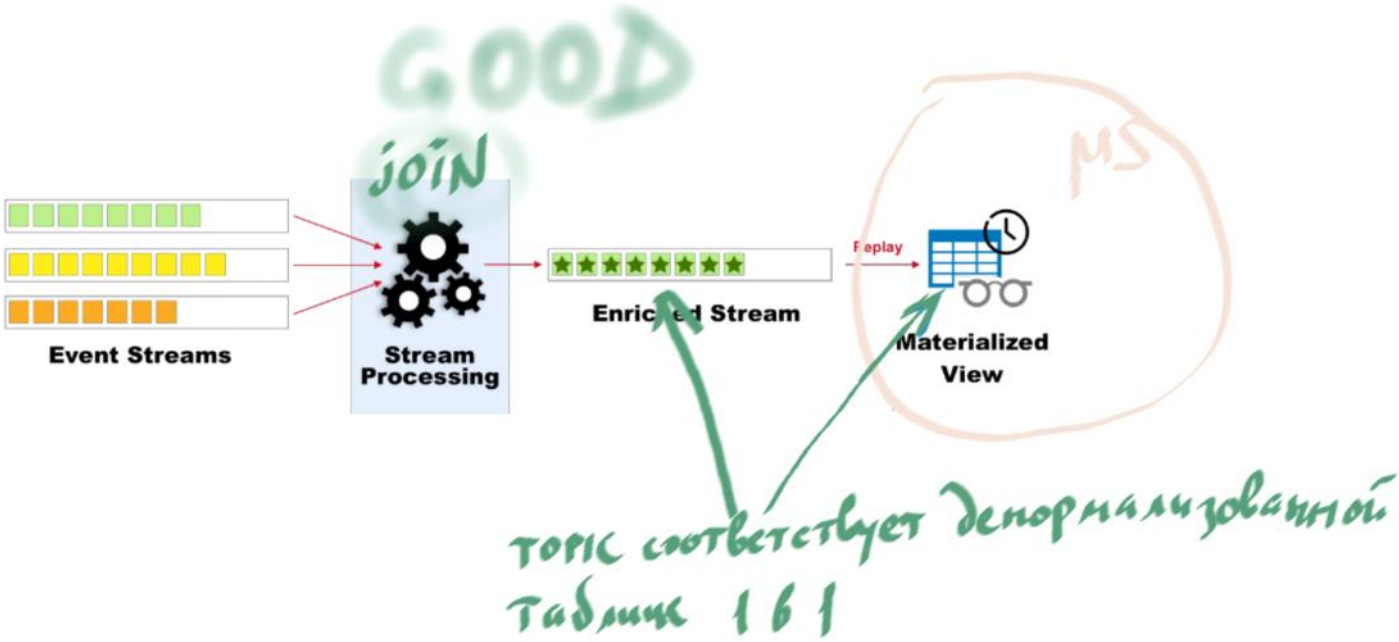


вариант: одна денормализованная таблица полностью соответствует одному compacted топику (который собирается из разных топиков в кафке)

- джойн данных происходит вовне микросервиса на кафке (KSQL)
- **для простоты** таблица заполняется (или при ошибке перезаполняется) с компактед топика

1) этот вариант можно сделать как на своей БД H2 **в памяти(но обычно в память такой объем даных не помещается)** или **ephimeral** local store
   - если удалить БД, то она автоматически наполнится из за того что у каждого инстанса новый consumer group id и auto-offset-reset=earliest
2) этот вариант можно сделать с kafka streams local RocksDB (но тогда бэкап ненужен .withLoggingDisabled() так как уже есть компактед топик)





открытый вопрос1: ребаланс консьюмеров (особенно временный ребаланс пока новый инстанс не поднялся)

- функционал доступен "из коробки" для kafka streams API

- вариант1) допустить временный ребаланс
  - при ребалансе топик и компактед топик подключатся к одному и тому же консьюмеру
  - но так как консьюмер группа уже выжрала из компактед топика сообщения то их нужно перезалить
- вариант2) не допускать временного ребаланса, а сделать static membership

открытый вопрос2: поиск по ключу в каком консьюмере находится клиент, чтобы перенаправить запрос на этого консьюмера

- функционал доступен "из коробки" для kafka streams API

* DB from NORMAL+COMPACTED topic

22 декабря 2020 г. 22:18

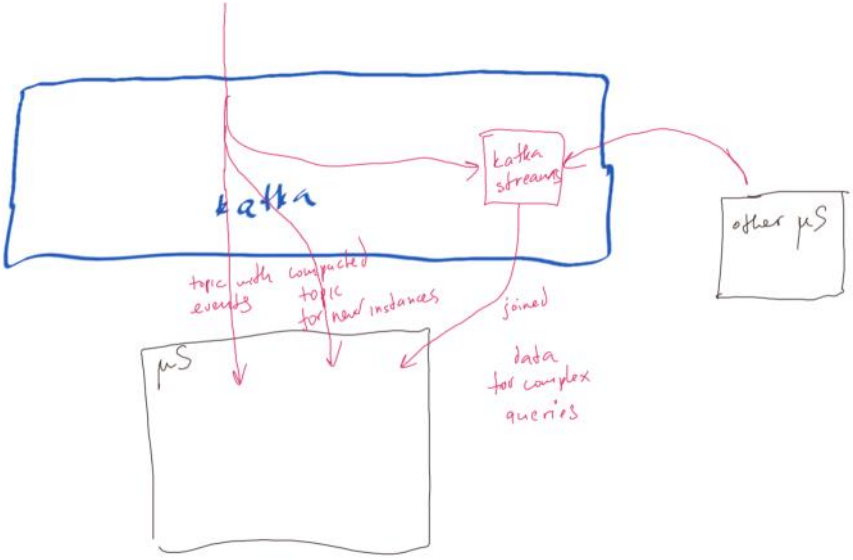
? как реализовать

The Latest-Versioned Pattern (как подпаттерн materialized views) в терминологии Confluent

- топик с обычными событиями используется для штатной работы сервиса
- compacted-topic используется для начальной загрузки или для восстановления из сбоев

- A useful pattern, when using event sourcing in this way, is to hold events twice: once in a retention-based topic and once in a compacted topic. The retention-based topic will be larger as it holds the 'version history' of your data. The compacted topic is just the 'latest' view, and will be smaller, so it's faster to load into a Memory Image or State Store.
- To keep the two topics in sync you can either dual write to them from your client (using a transaction to keep them atomic) o r, more cleanly, use Kafka Streams to copy one into the other. Later in this series we'll look at extending this pattern to create a whole variety of different types of vie ws.

.

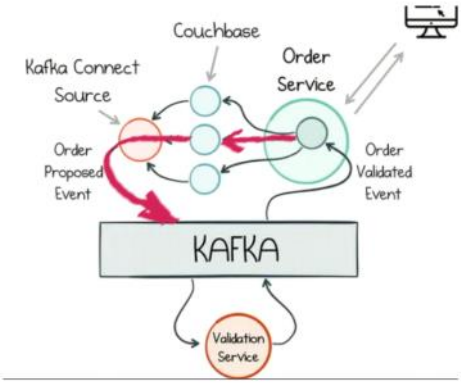


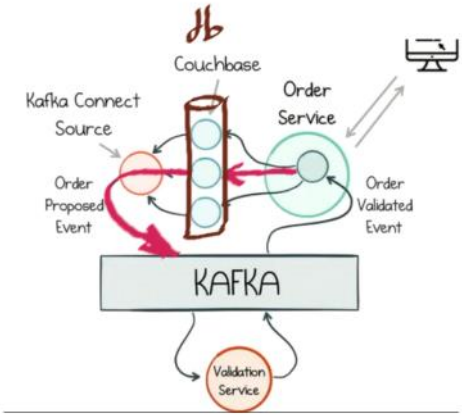
* write to kafka throught DB

8 декабря 2020 г. 21:47

нужно управлять транзакциями с помощью RDBMS

так как обычно обеспечивается eventual consistency, то чтобы обеспечить транзакционность нужно всетки использовать БД





Запись через базу данных в поток событий

Один из способов начать работу с событиями - это записать *через* таблицу базы данных в тему Kafka.

Самый надежный и эффективный способ сделать это - использовать технику, которая называется « Сбор измененных данных» (CDC). Большинство баз данных записывают каждую операцию вставки, обновления и удаления в журнал транзакций. Это служит «источником истины» для базы данных, и в случае ошибок состояние базы данных восстанавливается оттуда. Это означает, что мы также можем воссоздать состояние базы данных извне, скопировав журнал транзакций в Kafka через API-интерфейс Kafka Connect таким образом, чтобы события (операции вставки, обновления, удаления) были доступны для других служб.

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

Преимущество этого подхода, ориентированного на базу данных, заключается в том, что он обеспечивает точку согласованности: вы пишете через него в Kafka, что означает, что вы всегда можете читать свои собственные записи. Шаблон также менее подвержен ошибкам, чем запись в сторону, когда ваша служба записывает в базу данных, а затем записывает в журнал - шаблон, который требует распределенных транзакций для гарантированной точности.

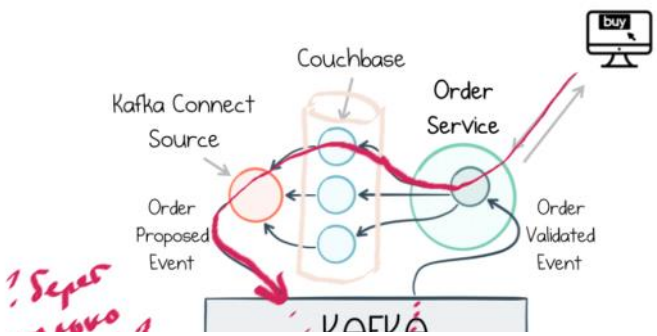
From <https://www.confluent.io/blog/messaging-single-source-truth/>

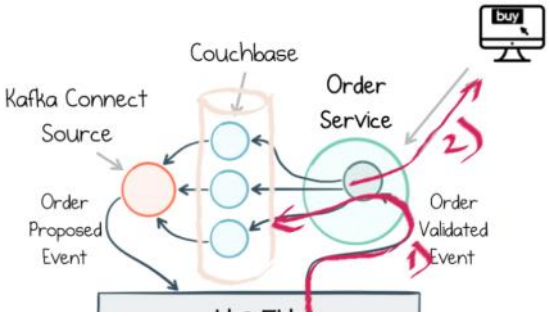
?? паттерн для больших БД хранилищ

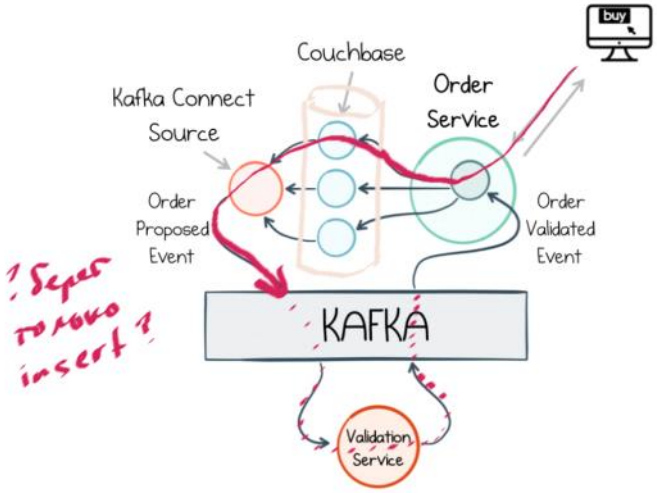
Writing Through a Database into an Event Stream (с использованием CDC kafka коннектора)

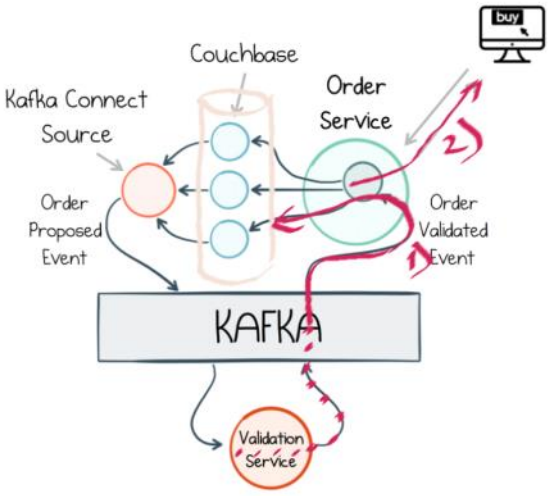
- The advantage of this 'database-fronted' approach is it provides a consistency point: you write through it into Kafka, meaning you can always read your own writes. т.е. для решения проблем отставания чтения своих же собственных данных в CQRS

- Преимущество этого подхода, ориентированного на базу данных, заключается в том, что он обеспечивает точку согласованности: вы пишете через него в Kafka, что означает, что вы всегда можете читать свои собственные записи. Шаблон также менее подвержен ошибкам, чем запись в сторону, когда ваша служба записывает в базу данных, а затем записывает в журнал - шаблон, который требует распределенных транзакций для гарантированной точности.
- Когда мы поговорим о потоковых подходах позже в этой серии, которые встраивают хранилище в наши сервисы, мы увидим, что они достигают эквивалентного результата.
- CDC доступен не для каждой базы данных, но экосистема растет.









чего существует вероятность, что пользователь сделает запись в журнал, затем попробует ее прочитать из производного представления и обнаружит, что эта запись еще не отражена в представлении для чтения

- Одним из решений данной проблемы является обновление представления для чтения одновременно с добавлением события в журнал.
- Для этого нужно, чтобы в транзакции искомые операции записи были объединены в неделимую единицу, независимо от того, нужно сохранить журнал событий и производить чтение в одной системе хранения или же требуется распределенная транзакция в разных системах.
- Вместо этого можно использовать технологию, описанную в пункте «Реализация линеаризуемого хранилища с помощью рассылки общей последовательности» подраздела «Рассылка общей последовательности»

можно использовать для этого CDC KAFKA CONECTOR

- Writing Through a Database into a Kafka Topic with Kafka Connect
- На рис. 7-9 служба заказов записывает заказы в базу данных. Записи преобразуются в поток событий с помощью API-интерфейса Kafka Connect. Это запускает последующую обработку, которая подтверждает заказ. Когда событие «Заказ подтвержден» возвращается в службу заказов, база данных обновляется с указанием окончательного состояния заказа, прежде чем вызов вернется к пользователю.

одна из целей паттерна "write throught DB" чтобы микросервис мог атомарно(те в одной локальной транзакции) обновлять свою базу данных и публиковать event/сообщение

- но в подходе где используется kafka streams можно просто использовать streams transactions и streams state store
- Чтобы события публиковались как часть транзакции, которая обновляет агрегат в базе данных, сервис должен использовать транзакционный обмен сообщениями. Такой механизм реализован в фреймворке Eventuate Tram, описанном в главе 3. Он вставляет события в таблицу OUTBOX в рамках ACID-транзакции, которая обновляет базу данных. После фиксации транзакции события, вставленные в таблицу OUTBOX, публикуются для брокера сообщений.

Pattern: Transactional outbox

- Использование таблицы базы данных в качестве очереди сообщений
- У сервиса, отправляющего сообщения, есть таблица OUTBOX (рис. 3.13). В рамках транзакции, которая создает, обновляет и удаляет бизнес-объекты, сервис шлет сообщения, вставляя их в эту таблицу. Поскольку это локальная ACID-транзакция, атомарность гарантируется
- Таблица OUTBOX играет роль временной очереди сообщений. Ретранслятор (MessageRelay) — это компонент, который читает таблицу OUTBOX и передает сообщения брокеру

- http://microservices.io/patterns/data/transactional-outbox.html

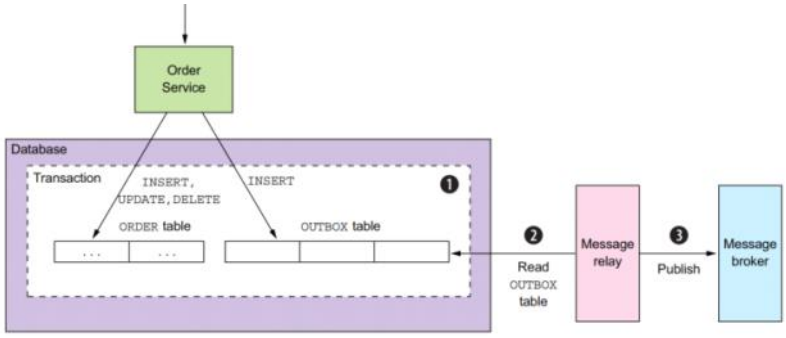


Figure 3.13 A service reliably publishes a message by inserting it into an `OUTBOX` table as part of the transaction that updates the database. The `Message Relay` reads the `OUTBOX` table and publishes the messages to a message broker.

POLLING PUBLISHER PATTERN

- https://microservices.io/patterns/data/polling-publisher.html
- Если приложение использует реляционную базу данных, сообщения, вставленные в таблицу OUTBOX, можно опубликовать очень простым способом: ретранслятору достаточно запросить из таблицы неопубликованные записи. Таблица периодически опрашивается
- Опрос базы данных — это простой подход, который неплохо работает в небольших масштабах. Его недостатком является то, что частое обращение к БД может оказаться затратным
- К тому же его можно использовать только с теми базами данных NoSQL, которые поддерживают соответствующие запросы. Это связано с тем, что вместо таблицы OUTBOX приложению приходится запрашивать бизнес-объекты, а это не всегда можно сделать эффективно.

lished messages. It periodically queries the table:

```
SELECT * FROM OUTBOX ORDERED BY ... ASC
```

Next, the `MessageRelay` publishes those messages to the message broker, sending one to its destination message channel. Finally, it deletes those messages from the `OUTBOX` table:

```
BEGIN
 DELETE FROM OUTBOX WHERE ID in (....)
COMMIT
```

~~плохое решение USING POLLING TO PUBLISH EVENTS на основе монотонно увеличивающегося event_id~~

- Самое сложное — определить, какие из событий новые
- Представьте, к примеру, что значения полей eventId увеличиваются монотонно. На первый взгляд было бы логично позволить издателю записывать последнее значение eventId, которое он обработал. После этого он смог бы извлечь новые события с помощью примерно такого запроса: SELECT * FROM EVENTS where event_id > ? ORDER BY event_id ASC
- Проблема этого подхода в том, что порядок фиксации транзакций может не совпасть с порядком, в котором они генерируют события. В итоге одно из событий может быть случайно пропущено издателем
- В этом сценарии транзакция A вставляет событие, у которого столбец EVENT_ID равен 1010. Затем транзакция B вставляет событие со столбцом EVENT_ID, равным 1020, и фиксируется. Теперь, если издатель выполнит запрос к таблице EVENTS, он найдет событие 1020. П озже, когда зафиксируется транзакция A, станет доступным событие 1010, но издатель его проигнорирует и событие так и не будет опубликовано

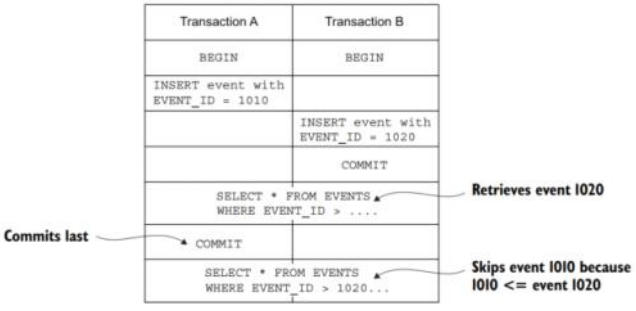


Figure 6.6 A scenario where an event is skipped because its transaction *A* commits after transaction *B*. Polling sees `eventId=1020` and then later skips `eventId=1010`.

USING POLLING TO PUBLISH EVENTS на основе дополнительного столбца, который отслеживает публикацию событий

- Одно из решений этой проблемы состоит в создании в таблице EVENTS дополнительного столбца, который отслеживает публикацию событий. В результате издатель использовал бы следующий процесс.

1. Find unpublished events by executing this SELECT statement: `SELECT * FROM EVENTS where PUBLISHED = 0 ORDER BY event_id ASC`.
2. Publish events to the message broker.
3. Mark the events as having been published: `UPDATE EVENTS SET PUBLISHED = 1 WHERE EVENT_ID in`.

TRANSACTION LOG TAILING PATTERN

- Менее тривиальное решение заключается в том, что ретранслятор отслеживает транзакционный журнал базы данных, который называют еще журналом фиксации. Каждое зафиксированное обновление, выполненное приложением, представлено в виде записи в журнале транзакций БД
- Анализатор журнала транзакций читает записи в транзакционном журнале. Каждую подходящую запись, которая соответствует добавлению сообщения, он преобразует в событие и публикует его для брокера. Этот подход годится для публикации сообщений, записанных в таблицу СУБД или добавленных в список записей в базе данных NoSQL https://microservices.io/patterns/data/transaction-log-tailing.html
- Это не самый простой подход, но работает он на удивление хорошо. Основная трудность состоит в том, что его реализация требует от разработчиков некоторых усилий
- There are a few examples of this approach in use:
  - Debezium (http://debezium.io)—An open source project that publishes database changes to the Apache Kafka message broker. Недостатком Debezium является то, что он предназначен для перехвата изменений на уровне БД и API для отправки и получения сообщений находятся вне его зоны ответственности.
  - LinkedIn Databus (https://github.com/linkedin/databus)—An open source project that mines the Oracle transaction log and publishes the changes as events. LinkedIn uses Databus to synchronize various derived data stores with the system of record.
  - DynamoDB streams (http://docs.aws.amazon.com/amazondynamodb/latest/developerguide/Streams.html)—DynamoDB streams contain the time-ordered sequence of changes (creates, updates, and deletes) made to the items in a DynamoDB table in the last 24 hours. An application can read those changes from the stream and, for example, publish them as events.
  - Eventuate Tram (https://github.com/eventuate-tram/eventuate-tram-core)—Your author's very own open source transaction messaging library that uses MySQL binlog protocol, Postgres WAL, or polling to read changes made to an OUTBOX table and publish them to Apache Kafka.
    - Транзакционный обмен сообщениями — публикует сообщения в рамках транзакции базы данных.
    - Обнаружение дубликатов — потребитель сообщений в Eventuate Tram обнаруживает и отклоняет повторяющиеся сообщения. Это очень важно, потому что, как обсуждалось в подразделе 3.3.6, потребитель должен обработать каждое сообщение ровно один раз.

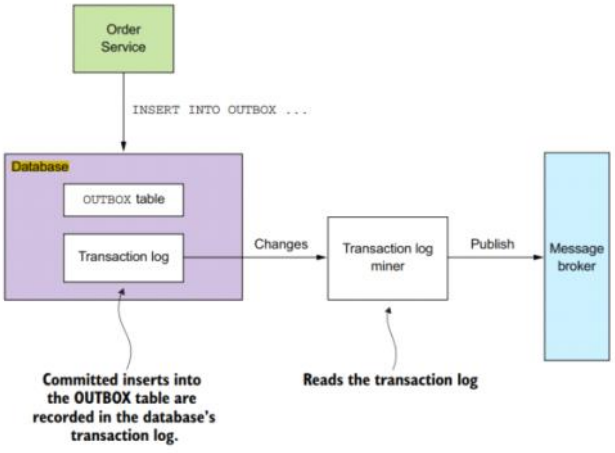


Figure 3.14 A service publishes messages inserted into the OUTBOX table by mining the database's transaction log.

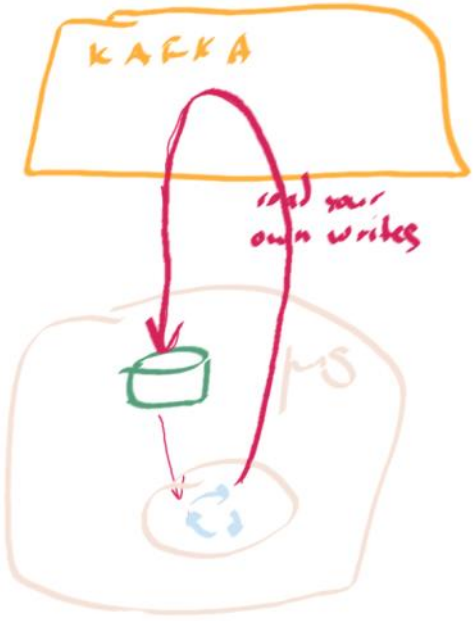
* write to kafka throught STATE STORE

2 января 2021 г. 15:16

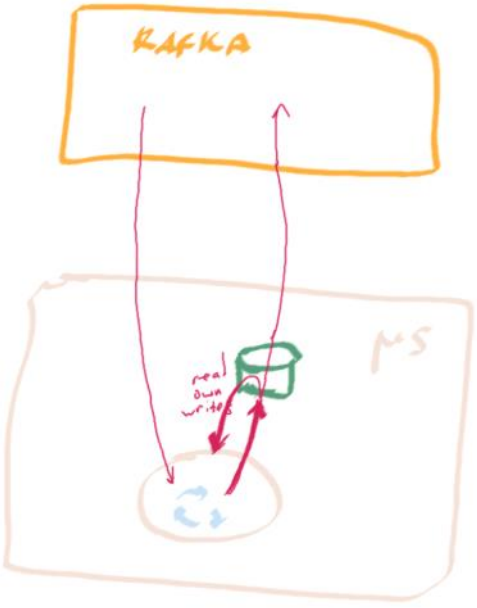
это то же самое что и шаблон " write to kafka throught DB" только в качестве БД выступает local state store

- База данных является локальной, что ускоряет доступ к ней.
- Поскольку хранилище состояний обернуто потоками Kafka Streams, оно может участвовать в транзакциях, поэтому события, публикуемые службой, и записи в хранилище состояний являются атомарными.
- Здесь меньше конфигурации, поскольку это единый API (без внешней базы данных и без коннектора CDC для настройки).

Вар 1)

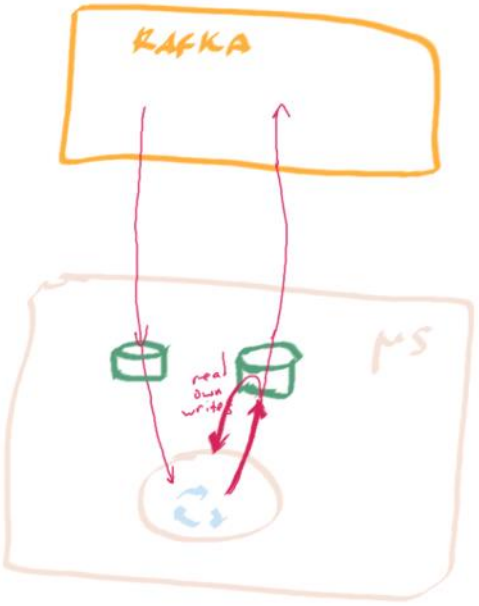


Вар 2)



??

Вар 3)



в варианте 1, если я хочу прочитать свои собстенные записи, то я должен блокирнуться пока они не дойдут то БД

- Клиент может получить семантику чтения-вашей-записи от любого узла, предоставив временную метку записи как часть своего запроса - обслуживающий узел, получающий такой запрос, сравнит желаемую временную метку со своей собственной индексной точкой и, при необходимости, задержит запрос до тех пор, пока он проиндексирован, по крайней мере, до этого времени, чтобы избежать обслуживания устаревших данных.

получение текущего состояния из журнала событий упрощает некоторые аспекты контроля конкурентного доступа.

- По большей части потребность в транзакциях с несколькими объектами связана с одним действием пользователя, требующим изменения данных в нескольких местах.
- С помощью источников событий можно построить событие таким образом, чтобы оно было самодостаточным, полностью описывая действие пользователя.
- Такое действие потребует всего одной записи в одном месте — а именно, добавления события в журнал, — и его легко сделать неделимым.
- Если журнал событий и состояние приложения одинаково секционированы (например, для обработки события для клиента в секции 3 в состоянии приложения требуется только обновление данной секции), то простой однопоточный потребитель журнала не нуждается в контроле конкурентного доступа для записи — по своей природе он способен обрабатывать события только последовательно (см. также подраздел «По-настоящему последовательное выполнение» раздела 7.3).
- Чтобы избежать неопределенности при конкурентном доступе, в журнале всем событиям секции присваиваются порядковые номера [24]. Если же событие касается нескольких секций, то требуется немного больше работы — об этом мы поговорим в главе 12.

* blocking read pattern(с помощью long polling)

4 января 2021 г. 19:09

onenote:///C:\Users\trans\Qsync\vova_from_onenote\tf_algonote_v1\SYSTEMDESIGN\KAFKA__EVENT%20DRIVEN.one#...%20сервис%201&section-id={7172B8D4-BDE4-4FFF-A011-59DE79779A94}&page-id={2C09DCAC-5EC0-4684-A569-01EDD18BABC4}&end

Collapsing CQRS with a Blocking Read для решения проблем отставания чтения своих же собственных данных в CQRS

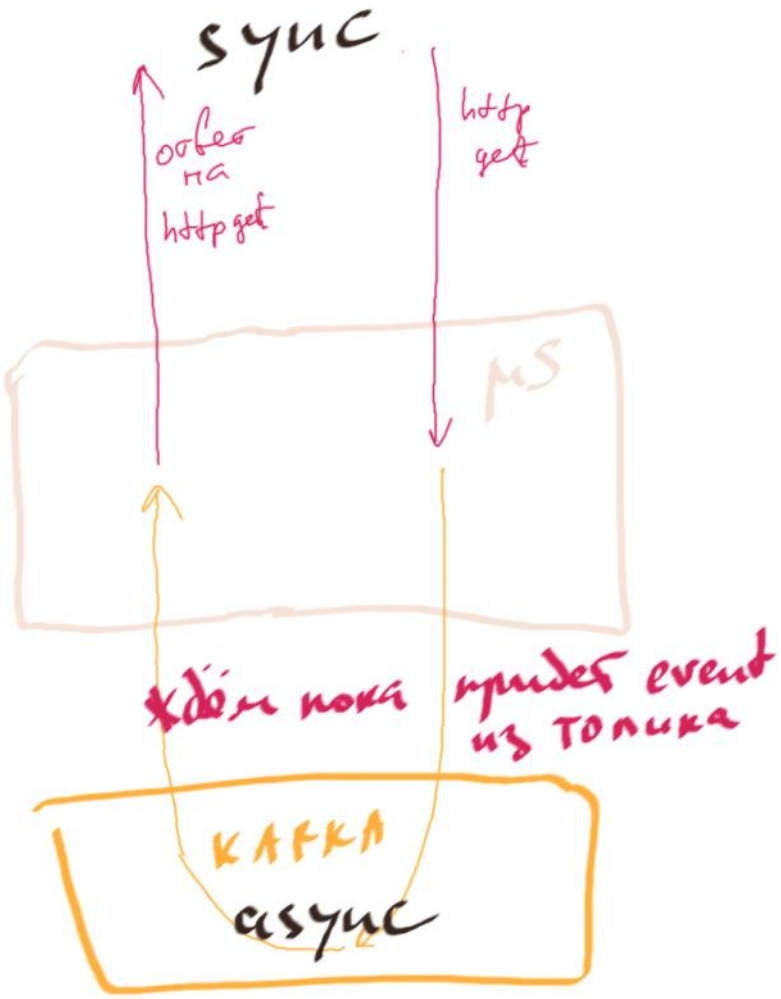
Collapsing CQRS with a Blocking Read" on page 142 in Chapter 15, there are strategies for addressing this problem.

Collapsing CQRS with a Blocking Read паттерн для обеспечения read your own writes

- Служба заказов реализует блокирующий HTTP GET, чтобы клиенты могли читать свои собственные записи.
- Одно из решений - заблокировать операцию GET до тех пор, пока не произойдет событие (или не пройдет заданный тайм-аут), сократив асинхронность шаблона CQRS, чтобы он выглядел синхронно для клиента. По сути, это метод длительного опроса . Служба заказов в примере кода реализует эту технику с использованием неблокирующего ввода-вывода

http long pooling как разновидность blocking read
те ответ на синхронный запрос блокируется (до тех пор пока сам микросервис не получит результат)

https://www.pubnub.com/blog/http-long-polling/



Orders Webservice

1. `POST /v1/orders` <- Create new order
2. `GET /v1/orders/<ID>` <- Get order details and status

используется long pooling для операции HTTP GET

неотвеченные ответы на HTTP GET будем хранить в мапе outstandingRequests

```
    //In a real implementation we would need to (a) support outstanding requests for the same Id/filter from
    // different users and (b) periodically purge old entries from this map.
    private Map<String, FilteredResponse<String, Order>> outstandingRequests = new ConcurrentHashMap<>();
```

при получении запроса HTTP GET добавим элемент в мапу outstandingRequests

```
/**
     * 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
     *
     * @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) Long timeout,
                               @Suspended final AsyncResponse asyncResponse) {
        setTimeout(timeout, asyncResponse);
        log.info("running GET on this node");
        try {
            Order order = ordersStore().get(id);
            if (order == null) {
                log.info("Delaying get as order not present for id " + id);
                outstandingRequests.put(id, new FilteredResponse<>(asyncResponse, (k, v) -> true));
            } else {
                asyncResponse.resume(order);
            }
        } catch (InvalidStateStoreException e) {
            //Store not ready so delay
            log.info("Delaying request for " + id + " because state store is not ready.");
            outstandingRequests.put(id, new FilteredResponse<>(asyncResponse, (k, v) -> true));
        }
    }
    /**
     * Perform a "Long-Poll" styled get. This method will attempt to get the order for the ID
     * blocking until the order has been validated 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
     *
     * @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}/validated")
    @Produces({MediaType.APPLICATION_JSON, MediaType.TEXT_PLAIN})
    public void getPostValidationWithTimeout(@PathParam("id") final String id,
                                             @QueryParam("timeout") @DefaultValue(CALL_TIMEOUT) Long timeout,
                                             @Suspended final AsyncResponse asyncResponse) {
        setTimeout(timeout, asyncResponse);
        log.info("running GET on this node");
        try {
            Order order = ordersStore().get(id);
            if (order == null || (order.getState() != OrderState.VALIDATED && order.getState() != OrderState.FAILED)) {
                log.info("Delaying get as a validated order not present for id " + id);
                outstandingRequests.put(id, new FilteredResponse<>(asyncResponse,
                        (k, v) -> (v.getState() == OrderState.VALIDATED || v.getState() == OrderState.FAILED)));
            } else {
                asyncResponse.resume(order);
            }
        } catch (InvalidStateStoreException e) {
            //Store not ready so delay
            log.info("Delaying request for " + id + " because state store is not ready.");
            outstandingRequests.put(id, new FilteredResponse<>(asyncResponse,
                    (k, v) -> (v.getState() == OrderState.VALIDATED || v.getState() == OrderState.FAILED)));
        }
    }
```

How do we use this table?

```
private ReadOnlyKeyValueStore<String, Order> ordersStore() {
    return streams.store(
                        ORDERS_STORE_NAME,
                        QueryableStoreTypes.keyValueStore());
}
```

How do we use this table?

```
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);
}
```

запустим работу стримсов

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;
}
```

если в топике появился новый event с заказом, то поймаем его

We are using KafkaStreams API to maintain a local state store

```
private StreamsBuilder createOrdersMaterializedView() {
    StreamsBuilder builder = new StreamsBuilder();
    builder.table(
            Topics.ORDERS.name(),
            Consumed.with(Topics.ORDERS.keySerde(),
                          Topics.ORDERS.valueSerde()),
            Materialized.as(ORDERS_STORE_NAME))
            .toStream()
            .foreach(this::maybeCompleteLongPollGet);
    return builder;
}
```

если есть неотвеченный ответ в outstandingRequests, то заполним его инфой из ивента топика заказов (неблокирущий метод)

// We just got a new update to our local state, which means an order was created or validated
// If there is an outstanding request for an update regarding that order
// AND the update satisfies the request (i.e. if the request is just for a "validated" response, don't return new orders)
// then unblock the caller

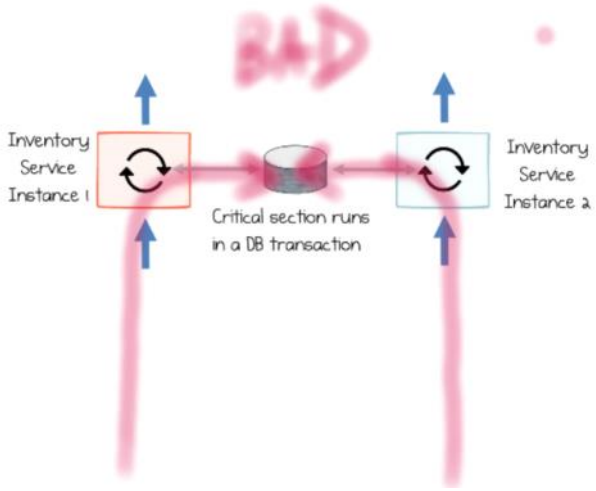
What's with the long poll?

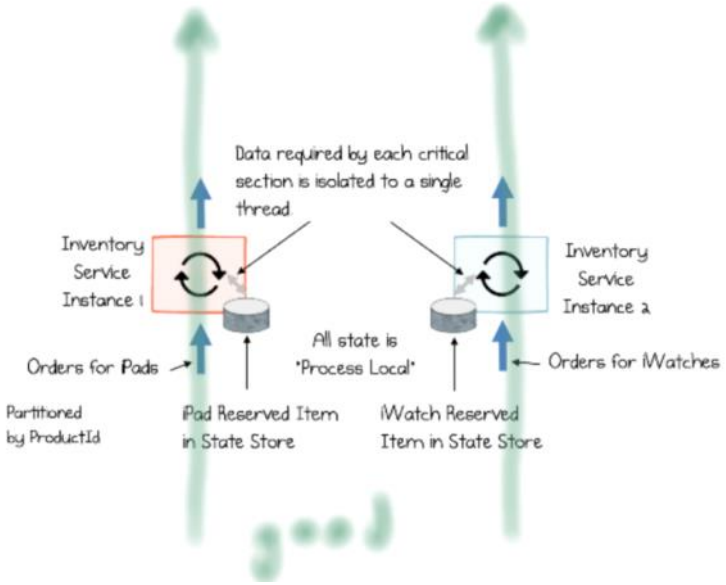
```
private void maybeCompleteLongPollGet(String id, Order
order) {
    FilteredResponse callback = outstandingRequests.get(id);

    if (callback != null &&
        callback.predicate.test(id, order)) {
            callback.asyncResponse.resume(order);
    }
}
```

* shared database ANTIpattern

5 января 2021 г. 10:05





snapshots to improve performance event spurcing

13 января 2021 г. 19:31

snapshots to improve performance

снапшот нужен для ускорения загрузки

- Время от времени состояние моментального снимка объекта домена может сохраняться вместе с порядковым номером события, на котором он основан; тогда восстановление может начаться с этого моментального снимка вместо того, чтобы возвращаться к началу времени.
- Приложение восстанавливает состояние агрегата, загружая его последний снимок и только те события, которые произошли с момента его создания.
  При восстановлении агрегата из снимка приложение сначала использует снимок для создания экземпляра агрегата, а затем последовательно применяет события
- Еще одна проблема, связанная с созданием представлений, состоит в том, что для обработки всех событий требуется все больше времени и ресурсов. В какой-то момент этот процесс становится слишком медленным и затратным. В качестве решения можно воспользоваться двухэтапным инкрементальным алгоритмом. Первый этап периодически вычисляет снимок экземпляров каждого агрегата с учетом предыдущего снимка и событий, произошедших с момента его создания. На втором этапе с помощью снимков и любых последующих событий создается представление

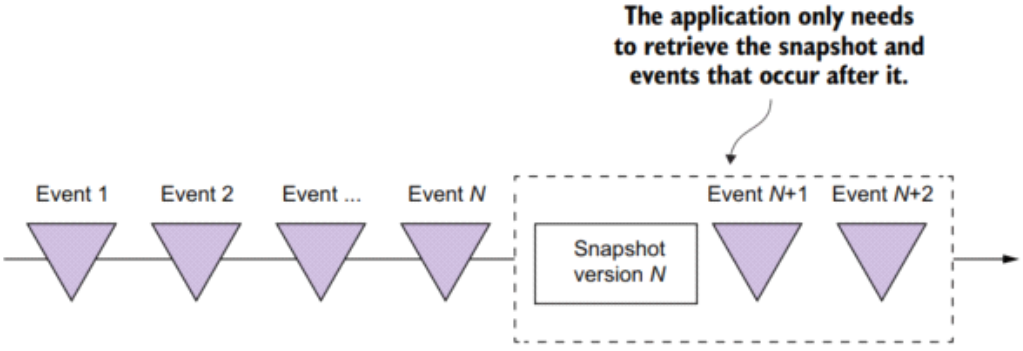


Figure 6.7 Using a snapshot improves performance by eliminating the need to load all events. An application only needs to load the snapshot and the events that occur after it.

снапшот сам по себе может содержать ошибки (если он получен из ошибочных событий)

- Проблема с этим подходом заключается в том, что изменения в логике предметной области (исправления ошибок) могут легко сделать снимки состояния недействительными, что необходимо признать и принять во внимание.

снапшот не имеет бизнес смысла (он нужен только для технических)

- Основная проблема заключается в том, что, хотя события имеют значение в бизнес-области, моментальный снимок не имеет - это просто проекция деталей реализации логики объекта домена.

фреймворк Eventuate Client, описанный в разделе 6.2, восстанавливает агрегат с помощью примерно такого кода

- Экземпляр агрегата восстанавливается из снимка, а не с помощью конструктора по умолчанию.
- Если у агрегата простая, легко сериализуемая структура, в качестве снимка может использоваться представление в формате JSON. Снимки более сложных агрегатов создаются с помощью шаблона «Хранитель»

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

Сервис Customer воссоздает агрегат Customer путем десериализации снимка в формате JSON с последующей загрузкой и применением событий с 104-го по 106-е.

EVENTS

event_id	event_type	entity_type	entity_id	event_data
...	...	...	...	...
103	...	Customer	101	{...}
104	Credit Reserved	Customer	101	{...}
105	Address Changed	Customer	101	{...}
106	Credit Reserved	Customer	101	{...}

SNAPSHOTS

event_id	entity_type	event_id	snapshot_data
...	...	...	...
103	Customer	101	{name: "..." , ...}
...	...	...	...
...	...	...	...

Figure 6.8 The `Customer Service` recreates the `Customer` by deserializing the snapshot's JSON and then loading and applying events #104 through #106.

антипаттерн local state store (Крис Ричардсон)

13 января 2021 г. 21:06

? с холодным резервированием БД на ленты, потому что БД невосстанавливается из кафки

- приложению нужно прочитать более старые события, заархивированные, скажем, в AWS S3. Это можно сделать с помощью масштабируемой технологии для хранения больших данных, такой как Apache Spark

Eventuate Local event store

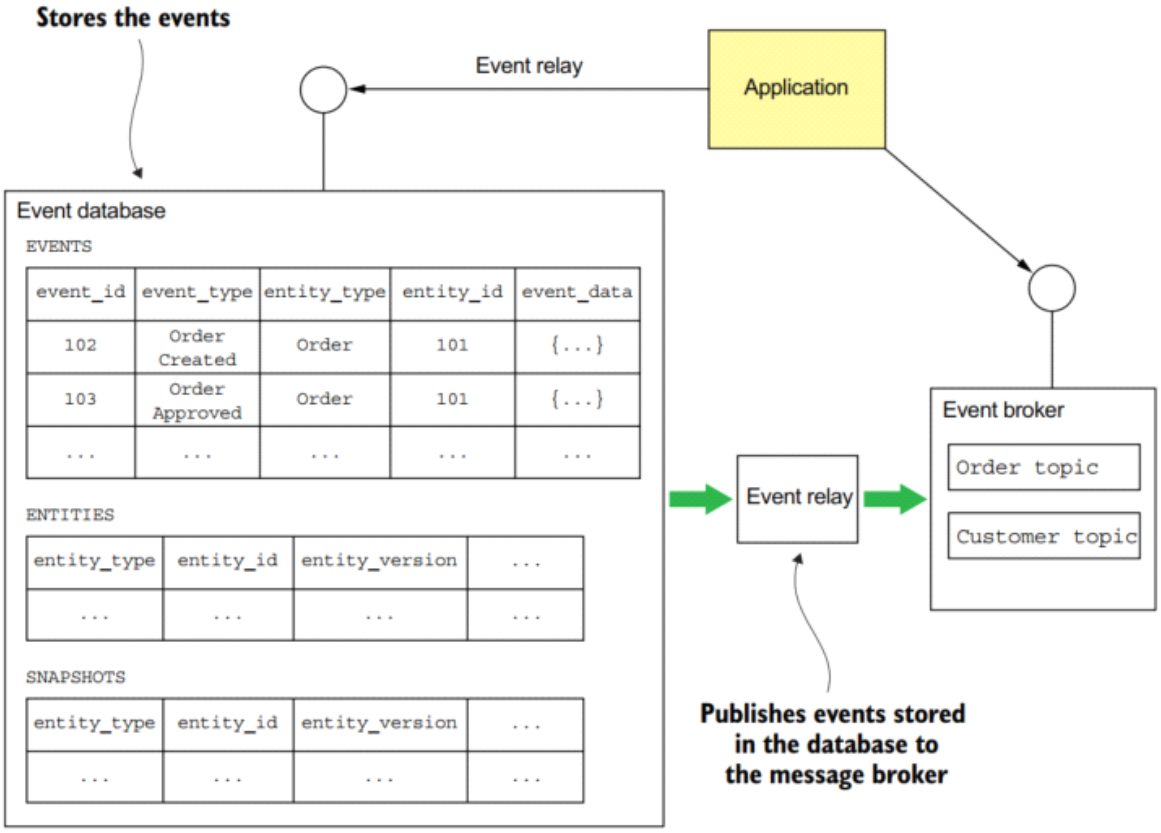


Figure 6.9 The architecture of Eventuate Local. It consists of an event database (such as MySQL) that stores the events, an event broker (like Apache Kafka) that delivers events to subscribers, and an event relay that publishes events stored in the event database to the event broker.

```
create table events (
  event_id varchar(1000) PRIMARY KEY,
  event_type varchar(1000),
  event_data varchar(1000) NOT NULL,
  entity_type VARCHAR(1000) NOT NULL,
  entity_id VARCHAR(1000) NOT NULL,
  triggering_event VARCHAR(1000)
);
```

The `triggering_event` column is used to detect duplicate events/messages. It stores the ID of the message/event whose processing generated this event.

The `entities` table stores the current version of each entity. It's used to implement optimistic locking. Here's the definition of this table:

```
create table entities (
  entity_type VARCHAR(1000),
  entity_id VARCHAR(1000),
  entity_version VARCHAR(1000) NOT NULL,
  PRIMARY KEY(entity_type, entity_id)
);
```

When an entity is created, a row is inserted into this table. Each time an entity is updated, the `entity_version` column is updated.

The `snapshots` table stores the snapshots of each entity. Here's the definition of this table:

```
create table snapshots (
  entity_type VARCHAR(1000),
  entity_id VARCHAR(1000),
  entity_version VARCHAR(1000),
  snapshot_type VARCHAR(1000) NOT NULL,
  snapshot_json VARCHAR(1000) NOT NULL,
  triggering_events VARCHAR(1000),
  PRIMARY KEY(entity_type, entity_id, entity_version)
)
```

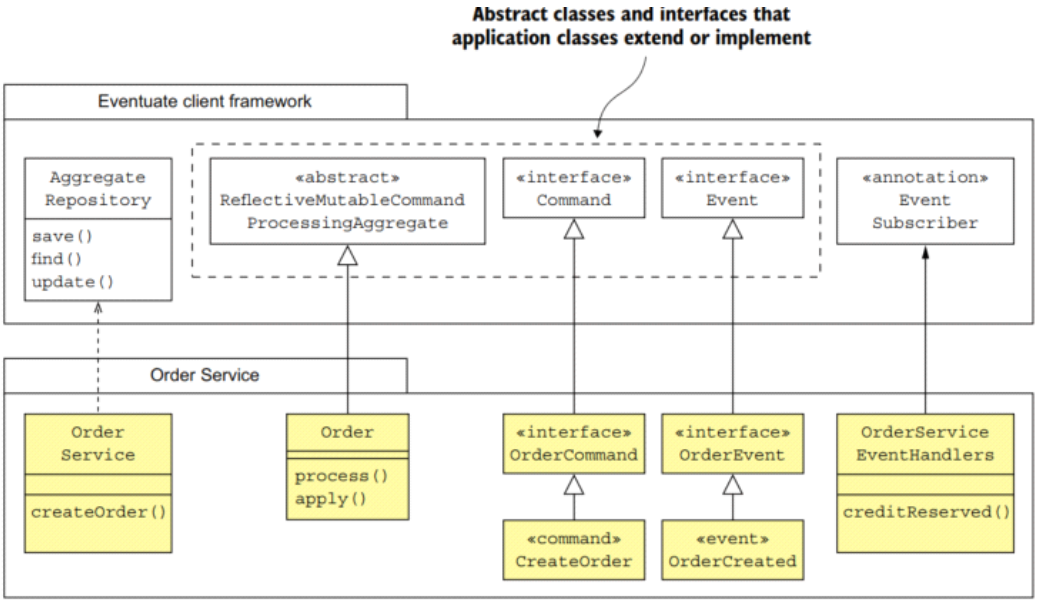


Figure 6.10 The main classes and interfaces provided by the Eventuate client framework for Java

Создание, поиск и обновление агрегатов с помощью класса AggregateRepository

- `save()`—Creates an aggregate
- `find()`—Finds an aggregate
- `update()`—Updates an aggregate

The `save()` and `update()` methods are particularly convenient because they encapsulate the boilerplate code required for creating and updating aggregates. For instance, `save()` takes a command object as a parameter and performs the following steps:

1. Instantiates the aggregate using its default constructor
2. Invokes `process()` to process the command

3. Applies the generated events by calling `apply()`
4. Saves the generated events in the event store

The `update()` method is similar. It has two parameters, an aggregate ID and a command, and performs the following steps:

1. Retrieves the aggregate from the event store
2. Invokes `process()` to process the command
3. Applies the generated events by calling `apply()`
4. Saves the generated events in the event store

Класс AggregateRepository в основном используют сервисы, которые создают и обновляют агрегаты в ответ на внешние запросы

Listing 6.4 `OrderService` uses an `AggregateRepository`

```
public class OrderService {
  private AggregateRepository<Order, OrderCommand> orderRepository;

  public OrderService(AggregateRepository<Order, OrderCommand> orderRepository)
  {
    this.orderRepository = orderRepository;
  }

  public EntityWithIdAndVersion<Order> createOrder(OrderDetails orderDetails) {
    return orderRepository.save(new CreateOrder(orderDetails));
  }
}
```

сохраняем состояние в трансформации

9 декабря 2020 г. 19:15

сохраняем состояние в трансформации

KStream
Transform / TransformValues

- Applies a Transformer to each record
- The Transformer leverages the low level Processor API

- The processor API is really advanced, not in scope for this course, but you can learn more about it here:
  - https://kafka.apache.org/0110/documentation/streams/developer-guide#streams_processor
  - http://docs.confluent.io/current/streams/developer-guide.html#streams-developer-guide-processor-api

KStream.transformValues — простейшая из функций с сохранением состояния

- Этот метод семантически не отличается от KStream.mapValues() с несколькими исключениями. Одно из отличий заключается в наличии у transformValues доступа к экземпляру интерфейса StateStore.
- Еще одно отличие — возможность планирования операций на выполнение через равные промежутки времени с помощью метода punctuate().

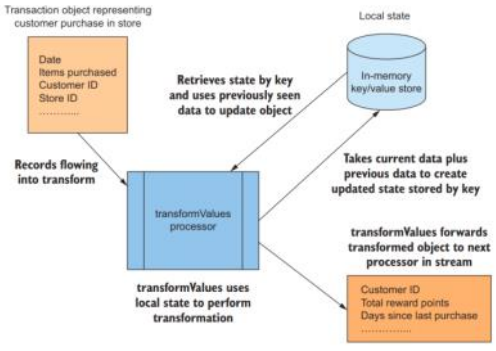


Figure 4.4 The `transformValues` processor uses information stored in local state to update incoming records. In this case, the customer ID is the key used to retrieve and store the state for a given record.

Метод KStream.transformValues() принимает на входе объект ValueTransformerSupplier<V, R>, служащий для создания экземпляра интерфейса ValueTransformer<V, R>

мы извлечем хранилище состояний, созданное при построении топологии обработки

Listing 4.2 `init()` method

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

private final String storeName;
private ProcessorContext context;

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

Instance variables

Sets a local reference to ProcessorContext

Retrieves the StateStore instance by storeName variable. storeName is set in the constructor.

внутри трансформации Сохранение суммарного количества бонусов для данного идентификатора покупателя в локальном хранилище состояния

Listing 4.3 Transforming `Purchase` using state

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

    if (accumulatedSoFar != null) {
        rewardAccumulator.addRewardPoints(accumulatedSoFar);
    }

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

    return rewardAccumulator;
}
```

Builds the Reward-Accumulator object from Purchase

Retrieves the latest count by customer ID

If an accumulated number exists, adds it to the current total

Stores the new total points in stateStore

Returns the new accumulated rewards points

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

- Важно размещать транзакции с одним идентификатором покупателя в одной
и той же секции для целей поиска записей по идентификатору в хранилище состояния

Секции/партициикузфкешещщ распределяются по объектам StreamTask, у каждого из которых есть свое хранилище состояния.)

Локальность данных критически важна для высокой производительности (те чтобы local state store был внутри микросервиса, а не в кафке)

- Хотя поиск по ключу обычно выполняется очень быстро при работе с большими объемами данных, задержка вследствие использования удаленных хранилищ оказывается узким местом.
- Сплошная линия изображает обращение к хранилищу данных в оперативной памяти, расположенному на том же сервере. Как вы можете видеть, обращаться за данными локально более выгодно с точки зрения производительности по сравнению с сетевым обращением к удаленной базе данных.
- Главное здесь не длительность задержки в пересчете на одну извлеченную запись, которая может быть минимальной. Суть в том, что потоковое приложение потенциально может обрабатывать миллионы или даже миллиарды записей, так что даже минимальная сетевая задержка, умноженная на такое число записей, может сыграть роковую роль.
- Локальность данных также означает локальность хранилища по отношению
ко всем обрабатывающим узлам и отсутствие совместного его использования различными процессами или потоками выполнения. Таким образом, сбой одного процесса не влияет на другие процессы или потоки выполнения при потоковой обработке.

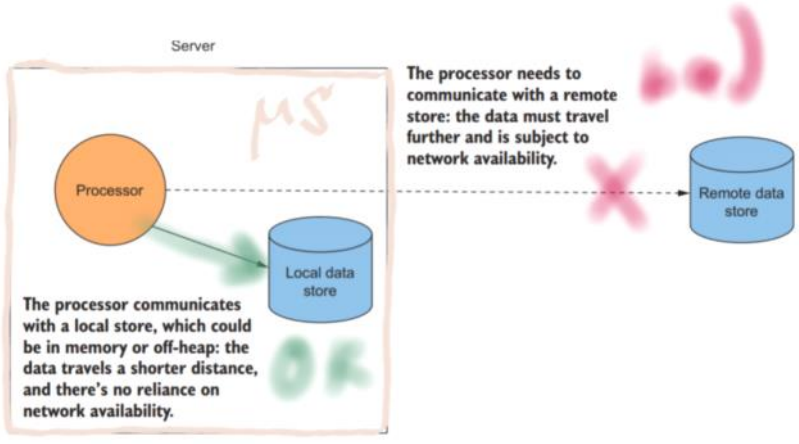
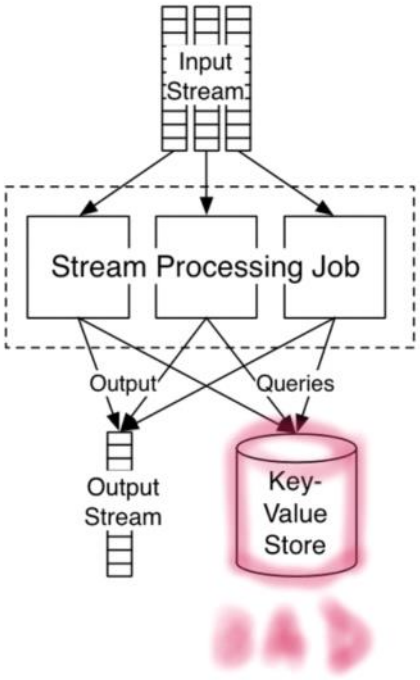
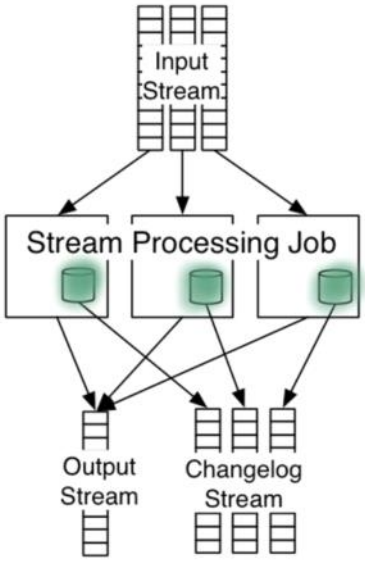


Figure 4.9 Data locality is necessary for stream processing.

- commitlog or changelog журнал фиксации или журнал изменений. В этот журнал записывается каждое изменение локального состояния задания.
- Когда задание изменяет свое локальное хранилище, оно регистрирует эти изменения в теме Kafka. Эту тему Kafka можно использова ть для воспроизведения изменений и восстановления локального состояния в случае сбоя компьютера и необходимости перезапуска процесса на новом хост





Эта модель state changes (изменений состояния) , поскольку сама по себе является своего рода потоком или журналом, довольно хороша.

- in fact, in many cases where the goal is to join together many different input streams, the changelog is also the only output of the job, and subscribed to by other downstream jobs

локальное состояние является фундаментальным примитивом в потоковой обработке.

Во-вторых, если локальное состояние для каждого процесса становится слишком большим, время восстановления для отказавшего задания замедляется.

- Для очень и очень больших наборов данных этот шаблон может не иметь смысла.

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 хранилище1
- 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).

восстановление после сбоя

- если сбои неизбежны то надо смириться с ними и Необходимо переместить акцент с предотвращения сбоев на быстрое восстановление после них или даже восстановление после перезапуска
- У каждого узла-обработчика — свое локальное хранилище данных, а топик для журнала изменений играет роль резервной копии хранилища состояния.
- Высокая стоимость резервного копирования хранилища состояния в топик смягчается за счет того, что KafkaProducer отправляет записи в пакетном режиме, и по умолчанию они кэшируются. Kafka Streams заносит записи в хранилище только при сбросе кэша на диск, так что сохраняется только последняя запись с заданным ключом
- Хранилища состояния Kafka Streams удовлетворяют требованию как локальности, так и отказоустойчивости. Они локальны по отношению к заданным узламобработчикам и не используются совместно различными процессами и потоками выполнения. Кроме того, для резервного копирования и быстрого восстановления хранилища состояния применяются топики.

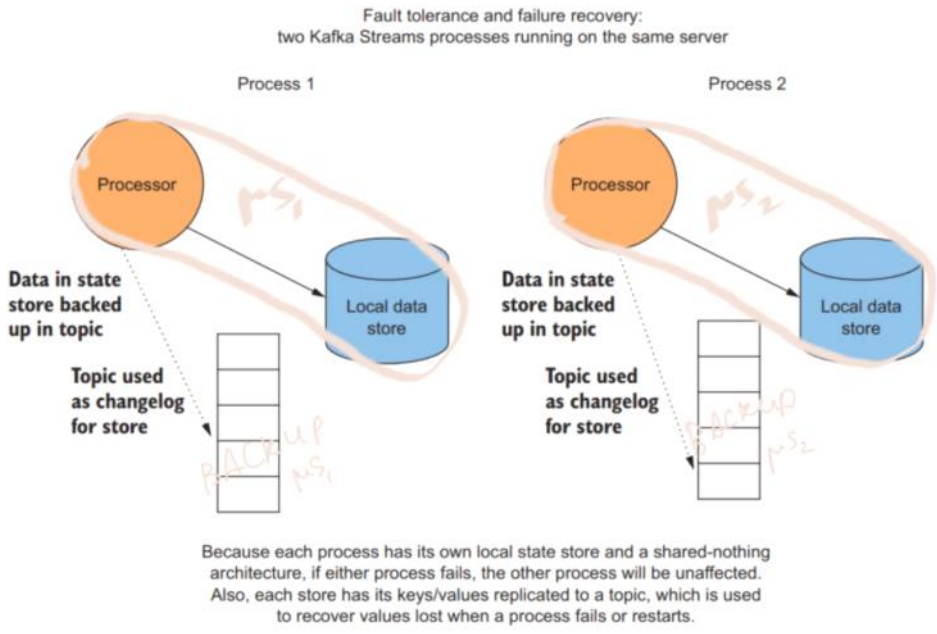


Figure 4.10 The ability to recover from failure is important for stream-processing applications. Kafka Streams persists data from the local in-memory stores to an internal topic, so when you resume operations after a failure or a restart, the data is repopulated.

Для добавления хранилища состояния достаточно создать экземпляр интерфейса StoreSupplier с помощью одного из статических фабричных методов класса Stores.

- Для настройки хранилища состояния под свои задачи существует два дополнительных класса: Materialized и StoreBuilder. Какой использовать — зависит от способа добавления хранилища в топологию. В случае высокоуровневого DSL обычно применяется класс Materialized, а при работе с низкоуровневым API узлов-обработчиков — StoreBuilder.
  - Сначала мы создаем объект StoreSupplier, который, в свою очередь, создает хранилище (в оперативной памяти) пар «ключ/значение»
  - Далее мы передаем этот StoreSupplier в качестве параметра при создании StoreBuilder, указывая также ключи в виде String и значения в виде Integer
  - Наконец, мы добавляем StateStore в топологию, передавая объект StoreBuilder в метод addStateStore
  - В результате мы получаем возможность обращаться к объекту для состояния в узлах-обработчиках по созданному выше имени rewardsStateStoreName.

Listing 4.7 Adding a state store

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

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

builder.addStateStore(storeBuilder);
```

Creates the StateStore supplier

Creates the StoreBuilder and specifies the key and value types

Adds the state store to the topology

другие статические фабричные методы для генерации поставщиков хранилищ:

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

```
Stores.inMemoryKeyValueStore
```

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

Во всех типах StateStoreSupplier журналирование/бэкапирование включено по умолчанию

- Во всех типах StateStoreSupplier журналирование включено по умолчанию. Журналирование в данном контексте означает использование журнала изменений в виде топика Kafka для резервного копирования значения из хранилища и обеспечения отказоустойчивости.
- Например, предположим, что произошел отказ одной из машин, на которых работает Kafka Streams. После восстановления сервера и перезапуска приложения Kafka Streams восстанавливается исходное содержимое хранилищ состояния этого экземпляра (последнее зафиксированное перед сбоем смещение в журнале изменений).
- Журналирование можно отключить при использовании фабрики Stores с помощью метода disableLogging(). Но при этом хранилища состояния теряют отказоустойчивость и способность восстанавливаться после аварийных сбоев, так что делать это без веских оснований не стоит.

disableLogging(). Но при этом хранилища состояния теряют отказоустойчивость и способность восстанавливаться после аварийных сбоев, так что делать это без веских оснований не стоит.

бэкап топики являются compacted

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

настройки бекап топиков

- Журналы изменений для хранилищ состояния можно настраивать с помощью метода withLoggingEnabled(Map <String, String> config). При этом можно использовать любые доступные в интерфейсе java.util.Map для топиков параметры конфигурации. Настройки журналов изменений для хранилищ состояния играют важную роль при создании приложений Kafka Streams.
- Но не забывайте, что создавать вручную топики журналов изменений не нужно — Kafka Streams делает это за вас.

- Рассмотрим сначала, как настроить топик журнала изменений так, чтобы установить объем сохраняемой информации равным 10 Гбайт и период сохранения — двум дням

Listing 4.8 Setting changelog properties

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

// to use with a StoreBuilder
storeBuilder.withLoggingEnabled(changeLogConfigs);

// to use with Materialized
Materialized.as(Stores.inMemoryKeyValueStore("foo")
                      .withLoggingEnabled(changeLogConfigs));
```

- в сжатых топиках применяется другой подход к очистке. Вместо удаления сегментов журнала время от времени сегменты журналов сжимаются (compacted): для каждого ключа оставляется только последняя запись, а более старые записи с тем же ключом удаляются. По умолчанию Kafka Streams создает топики журналов изменений со стратегией очистки compact.
- Но если в вашем топике журнала изменений множество уникальных ключей, то сжатия может оказаться недостаточно, так как размер сегмента журнала будет продолжать расти. Существует простое решение этой проблемы — достаточно задать стратегию очистки delete и compact .
  Теперь размер вашего топика журнала изменений останется в разумных пределах даже с уникальными ключами.

Listing 4.9 Setting a cleanup policy

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

примеры из SAMZA

https://www.oreilly.com/content/why-local-state-is-a-fundamental-primitive-in-stream-processing/
http://samza.incubator.apache.org/learn/documentation/0.7.0/container/state-management.html
https://www.confluent.io/blog/unifying-stream-processing-and-interactive-queries-in-apache-kafka/

samza+kafka+python

Stream processing in python with Apache Samza and Beam

changelogs

В случае сбоя приложения Kafka Streams или его перезапуска вручную хранилище состояния может восстановиться из локальных файлов состояния

В некоторых случаях, однако, может понадобиться полное восстановление хранилища состояния из журнала изменений

- например при работе приложения Kafka Streams в среде без сохранения состояния, вроде Mesos, или при серьезном сбое с потерей файлов на локальном диске.
- В зависимости от объема требующих восстановления данных такой процесс восстановления может потребовать немалого времени.

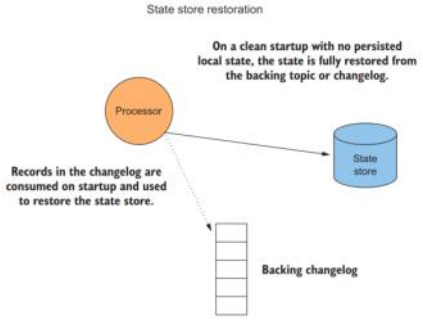


Figure 7.14 Restoring a state store from clean start/recovery

как узнать когда восстановление из топика закончится

- Во время этого периода восстановления все хранилища состояния, открытые вами для выполнения запросов, недоступны, так что хотелось бы понимать, сколько этот процесс восстановления займет времени и как он продвигается. Кроме того, при наличии пользовательского хранилища состояния будет не лишним получить оповещения о начале и завершении восстановления, чтобы выполнить нужные на этих этапах задачи.

- Интерфейс StateRestoreListener аналогично StateListener дает возможность оповещения о происходящем внутри приложения. Интерфейс StateRestoreListener включает три метода: onRestoreStart, onBatchRestored и onRestoreEnd. Для задания глобального прослушивателя восстановления используется метод KafkaStreams.setGlobalRestoreListener

Listing 7.6 A logging restore listener

```
public class LoggingStateRestoreListener implements StateRestoreListener {

    private static final Logger LOG =
➥ LoggerFactory.getLogger(LoggingStateRestoreListener.class);
    private final Map<TopicPartition, Long> totalToRestore =
➥ new ConcurrentHashMap<>();                                          <- Creates Concurrent-HashMap instances for keeping track of restore progress
    private final Map<TopicPartition, Long> restoredSoFar =
➥ new ConcurrentHashMap<>();                                          <-

    @Override
    public void onRestoreStart(TopicPartition topicPartition,
➥ String store, long start, long end) {                               <- Stores the total amount to restore for the given TopicPartition
        long toRestore = end - start;
        totalToRestore.put(topicPartition, toRestore);                 <-
        LOG.info("Starting restoration for {} on topic-partition {}
➥ total to restore {}", store, topicPartition, toRestore);
    }

    // other methods left out for clarity covered below
}
```

код, выполняющий собственно восстановление каждого пакета записей

- Для чтения из топика журнала изменений в процессе восстановления используется внутренний потребитель, поэтому приложение восстанавливает записи пакетами, соответствующими вызовам метода consumer.poll(). Вследствие этого максимальный размер любого пакета равен значению параметра конфигурации max.poll.records.
- После загрузки процессом восстановления последнего пакета в хранилище состояния вызывается метод onBatchRestored. Во-первых, мы прибавляем размер текущего пакета к накопленному количеству восстановленных записей. Далее подсчитываем степень

завершения восстановления (в процентах) и выводим в журнал результат. И наконец, сохраняем ранее вычисленное новое общее количество записей.

Listing 7.7 Handling onBatchRestored

```
@Override
public void onBatchRestored(TopicPartition topicPartition,
➥ String store, long start, long batchCompleted) {
  NumberFormat formatter = new DecimalFormat("#.##");

  long currentProgress = batchCompleted +
➥ restoredSoFar.getOrDefault(topicPartition, 0L);
  double percentComplete =
➥ (double) currentProgress / totalToRestore.get(topicPartition);

  LOG.info("Completed {} for {}% of total restoration for {} on {}",
           batchCompleted,
➥ formatter.format(percentComplete * 100.00),
➥ store, topicPartition);
➥ restoredSoFar.put(topicPartition, currentProgress);
}
```

Calculates the total number of records restored

Determines the percentage of restoration completed

Stores the number of records restored so far

Logs the percent restored

Метод, вызываемый по завершении процесса восстановления

Listing 7.8 Method called when restoration is completed

```
@Override
public void onRestoreEnd(TopicPartition topicPartition,
➥ String store, long totalRestored) {
   LOG.info("Restoration completed for {} on
➥ topic-partition {}", store, topicPartition);
   restoredSoFar.put(topicPartition, 0L);
}
```

Keeps track of restore progress for a TopicPartition

пример

var 1) bad

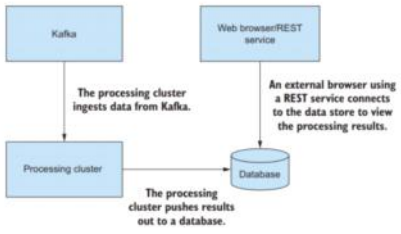


Figure 9.4 Architecture of applications viewing processed data prior to Kafka Streams

var 2) bad

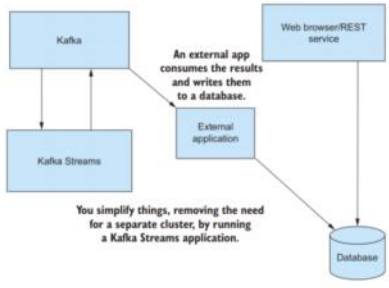


Figure 9.5 Architecture with Kafka Streams and state added

var 3) good

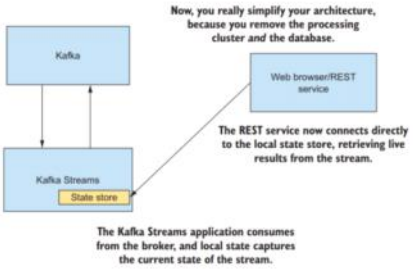


Figure 9.6 Architecture using interactive queries

Kafka Streams предоставляет доступ к хранилищам состояния с помощью метода KafkaStreams.store

- Чтобы интерактивные запросы могли работать, Kafka Streams предоставляет доступ (только для чтения) к хранилищам состояния.
- Важно понимать, что хранилища состояния, хотя для чтения и доступны, никак не обновляются и не модифицируются.
- Kafka Streams предоставляет доступ к хранилищам состояния с помощью метода KafkaStreams.store

Here's how this method works:

```
ReadOnlyWindowStore readOnlyStore =
➥ kafkaStreams.store(storeName, QueryableStoreTypes.windowStore());
```

This example retrieves a `WindowStore`. In addition, `QueryableStoreTypes` provides two other types:

- `QueryableStoreTypes.sessionStore()`
- `QueryableStoreTypes.keyValueStore()`

Kafka Streams assigns a state store per task

- Не забывайте, что Kafka назначает по хранилищу состояния для каждой задачи и приложение Kafka Streams может состоять из нескольких экземпляров, лишь бы у них был один идентификатор приложения.
- Кроме того, эти экземпляры вовсе не обязаны располагаться на одной машине. Следовательно, может так случиться, что хранилище состояния, к которому производится запрос, содержит только некое подмножество ключей; а другие хранилища состояния (с тем же названием, расположенные на других машинах) содержат другое подмножество ключей.

Как вы можете видеть, экземпляр А обрабатывает записи из секции 0, а экземпляр Б — записи из секции 1.

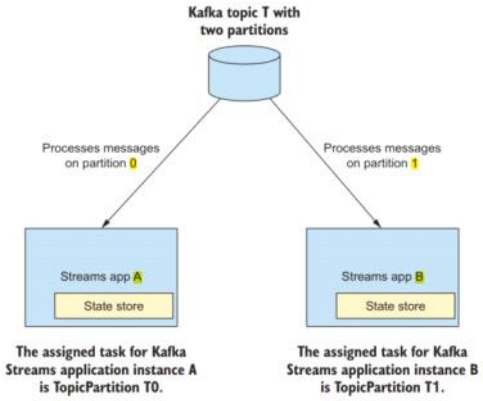


Figure 9.7 Task and state store distribution