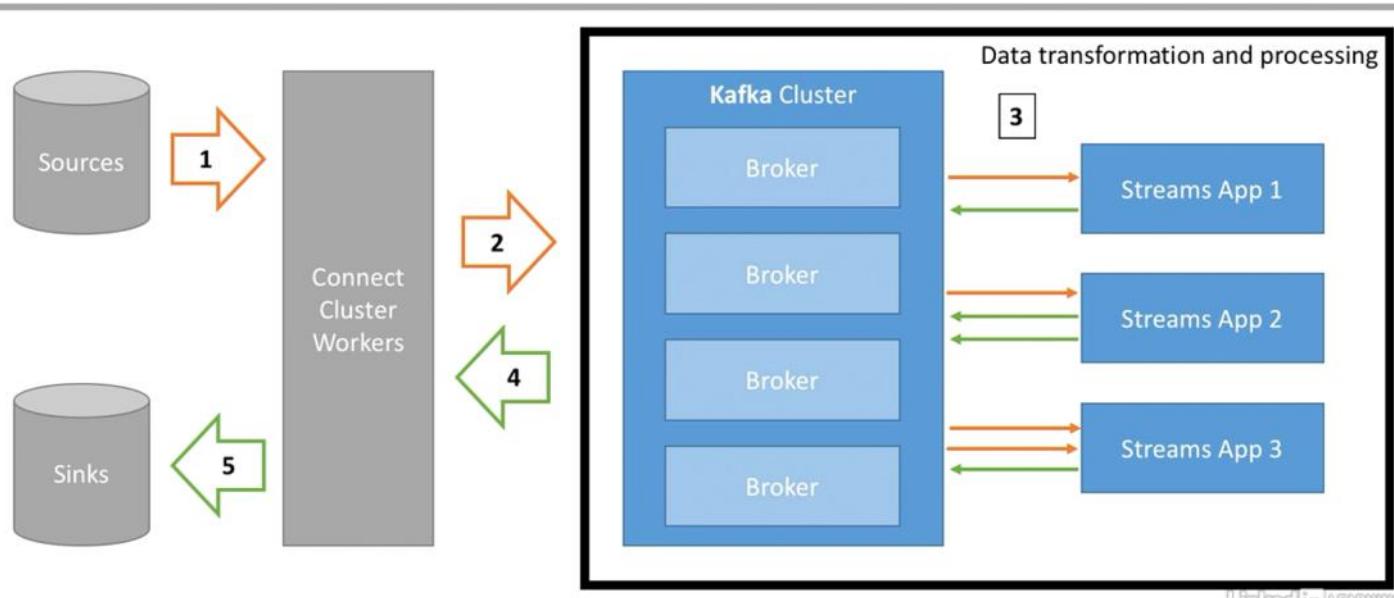


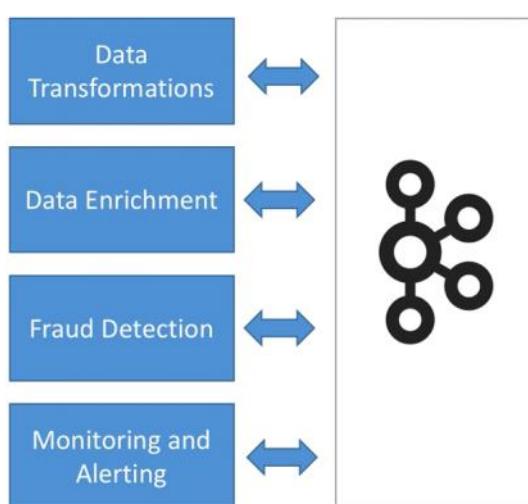
kafka streams это **JVM-only технология (только Java и Scala)**

Kafka Streams Architecture Design



? не нужен отдельный кластер

- Easy **data processing and transformation library** within Kafka



- Standard Java Application
- No need to create a separate cluster
- Highly scalable, elastic and fault tolerant
- Exactly Once Capabilities
- One record at a time processing (no batching)
- Works for any application size

Kafka Streams vs Spark Streaming, NiFi, Flink

- Micro batch vs per data streaming
- Cluster required vs no cluster required
- Scales easily by just adding java processes (no re-configuration required)
- Exactly Once semantics (vs at least once for Spark)
- All code-based
- <https://www.quora.com/What-are-the-differences-and-similarities-between-Kafka-and-Spark-Streaming>

application id это тоже самое что и консьюмер групп id (== consumergroup id)

```
config.put(StreamsConfig.APPLICATION_ID_CONFIG, "wordcount-application");
```

WordCount Streams App Properties



- A stream application, when communicating to Kafka, is leveraging the **Consumer** and **Producer** API.
- Therefore all the configurations we learned before are still applicable.
 - `bootstrap.servers`: need to connect to kafka (usually port 9092)
 - `auto.offset.reset.config`: set to `earliest` to consume the topic from start
 - `application.id`: specific to Streams application, will be used for
 - Consumer `group.id` = `application.id` (most important one to remember)
 - Default `client.id` prefix
 - Prefix to internal changelog topics
 - `default.[key|value].serde` (for Serialization and Deserialization of data)

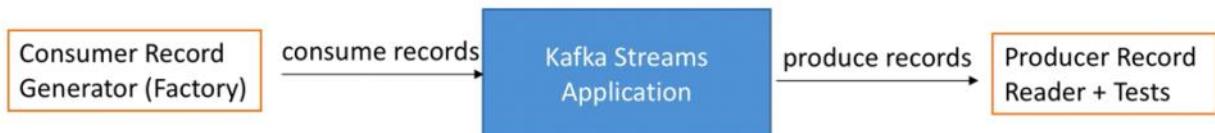
Kafka Streams testing!



- When Running a Kafka Streams Application:



- When Testing a Kafka Streams application

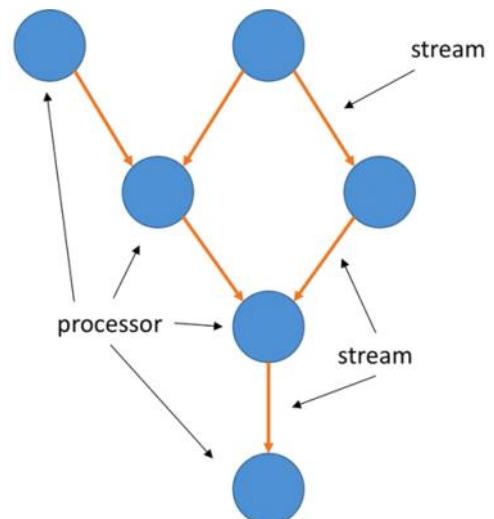


stream / table / topology

9 декабря 2020 г. 12:19

СТРИМ - это последовательность **немутабельных** объектов (похож на лог)

- A **stream** is a sequence of immutable data records, that fully ordered, can be replayed, and is fault tolerant (think of a Kafka Topic as a parallel)
- A **stream processor** is a node in the processor topology (graph). It transforms incoming streams, record by record, and may create a new stream from it.
- A **topology** is a graph of processors chained together by streams

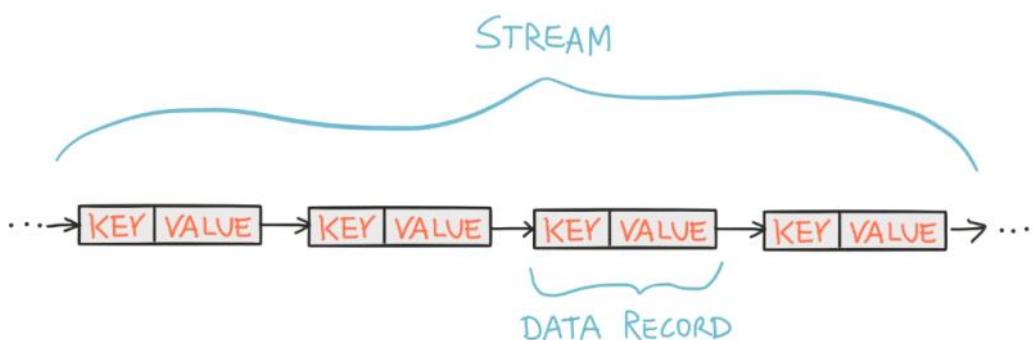


- A stream in Kafka is the full history from the start of time
- Messages in a stream
 - Constantly arriving & being added to the topic
 - Are independent and form a never-ending sequence
 - Arrive in a time ordered manner in the stream
 - Do not have any relation with each other
 - Can be processed independently

стрим - это **flow of records**

What is a Stream?

- Think of a stream as an unbounded, continuous real-time **flow of records**
 - You don't need to explicitly request new records, you just receive them
- Records are key-value pairs



топик -> стримы -> топик

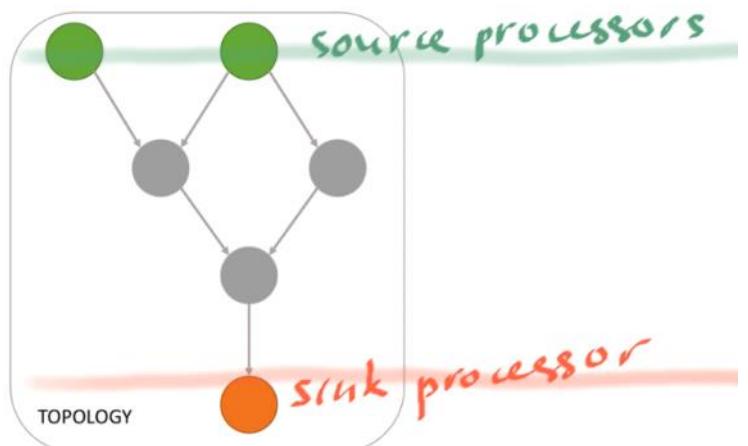
изначально задумывалось что начальной и конечной точками стрима являются кафка топик

TOPOLOGY - сама прога (те весь конвеер по обработке целиком)

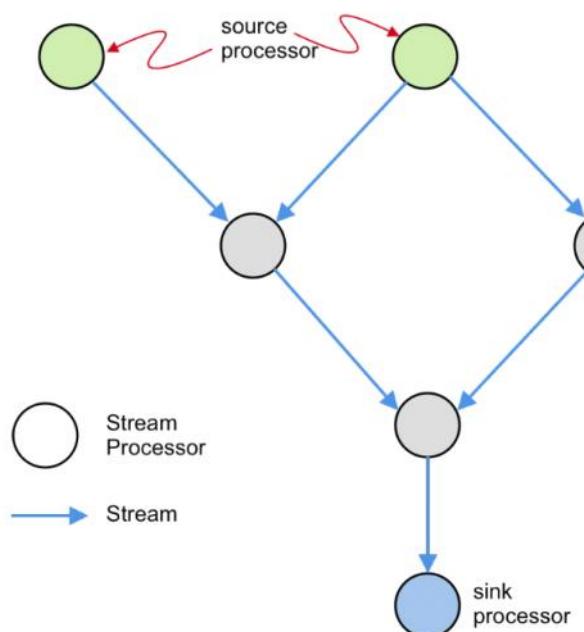
- топология - это граф обработки
- Топология — способ организации частей системы и соединения их друг с другом. Говоря о топологии Kafka Streams, я подразумеваю преобразование данных посредством их пропускания через один или несколько узлов обработки
- Kafka Streams дает возможность путем объединения узлов-обработчиков элегантно создавать сложные информационные потоки.

- Reminder: The topology represents all the streams and processors of your Streams application

- a **Source processor** is a special processor that takes its data directly from a Kafka Topic. It has no predecessors in a topology, and doesn't transform the data.
- a **Sink processor** is processor that does not have children, it sends the stream data directly to a Kafka topic



Stream Processor and Topology



- A *stream processor* transforms data in a stream
- A *processor topology* defines the data flow through the stream processors

- A stream processor transforms data in a stream
- A processor topology defines the data flow through the stream processors

A common misconception with the processor topology is that the "stream processors" are different systems. This is incorrect!

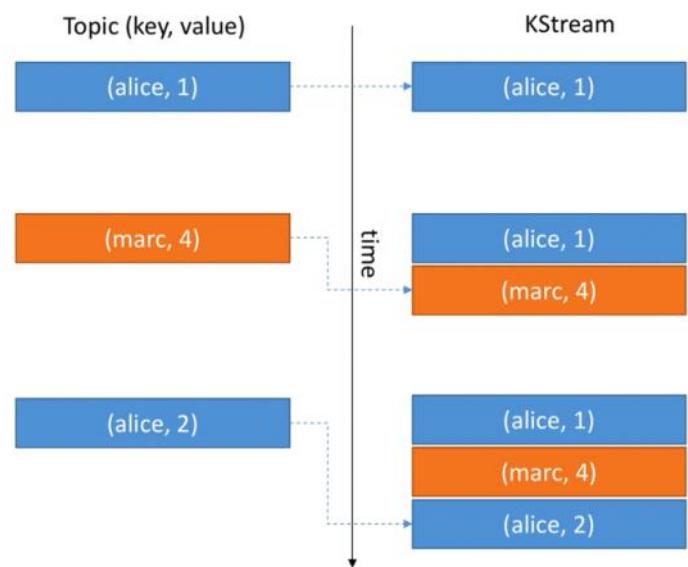
Correct definition:

A stream processor is a node in the processor topology that represents a single processing step. With the **Processor API**, you can define arbitrary stream processors that processes one received record at a time, and connect these processors with their associated state stores to compose the processor topology.

KStreams



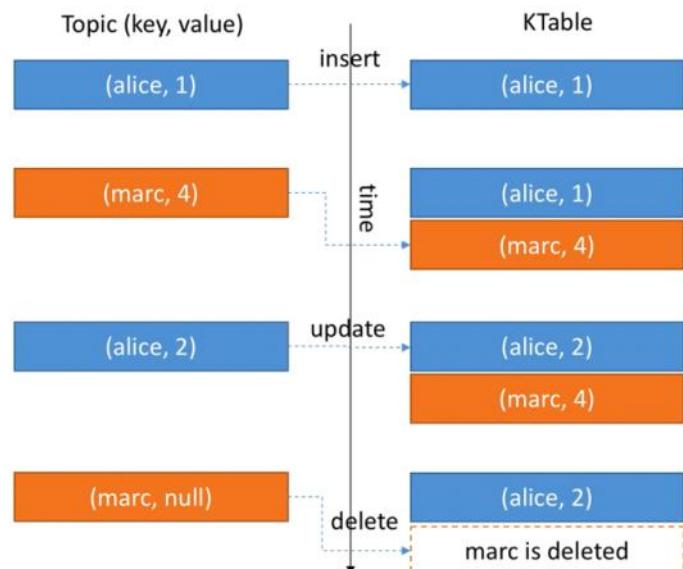
- All **inserts**
- Similar to a log
- Infinite
- Unbounded data streams



KTables



- All **upserts** on non null values
- Deletes on null values
- Similar to a table
- Parallel with log compacted topics



ktable в отличие от стрима **делает UPSERT(update or insert)** записи,

или даже ее удаляет если value==null

- также table должен быть только из одного топика (a stream можно сделать из нескольких топиков)

можно думать о таблице как о топике с бесконечным retention

- Kafka Streams развивает ту же концепцию на шаг впереди для управления целыми таблицами. Таблицы - это локальное проявление полной темы - обычно сжатой - хранящейся в хранилище состояний по ключу. (Вы также можете думать о них как о потоках с неограниченным сроком хранения.) В контексте микросервисов такие таблицы часто используются для обогащения

При использовании KTable хорошо то, что он ведет себя как таблица в базе данных.

Поэтому, когда мы присоединяем поток заказов к KTable клиентов, нам не нужно беспокоиться о сроках хранения, окнах или любой другой сложности. Если запись о клиенте существует, соединение будет работать.

When to use KStream vs KTable ?



- **KStream** reading from a topic that's not compacted
- **KTable** reading from a topic that's log-compacted (aggregations)
- **KStream** if new data is partial information / transactional
- **KTable** more if you need a structure that's like a "database table", where every update is self sufficient (think – total bank balance)

KStream & KTable Reading from Kafka



- You can read a topic as a KStream, a KTable or a GlobalKTable

```
KStream<String, Long> wordCounts = builder.stream(  
    Serdes.String(), /* key serde */  
    Serdes.Long(), /* value serde */  
    "word-counts-input-topic" /* input topic */);
```

```
KTable<String, Long> wordCounts = builder.table(  
    Serdes.String(), /* key serde */  
    Serdes.Long(), /* value serde */  
    "word-counts-input-topic" /* input topic */);
```

```
GlobalKTable<String, Long> wordCounts = builder.globalTable(  
    Serdes.String(), /* key serde */  
    Serdes.Long(), /* value serde */  
    "word-counts-input-topic" /* input topic */);
```

through() - можно записать в топик и не закрывать стрим (а получить ссылку на новый стрим)

round-trip (produce/consume) to Kafka, and the data read back from Kafka feeds into the next operator in the topology

KStream & KTable Writing to Kafka



- You can write any KStream or KTable back to Kafka
- If you write a KTable back to Kafka, think about creating a log compacted topic!
- To: Terminal operation – write the records to a topic

```
stream.to("my-stream-output-topic");
table.to("my-table-output-topic");
```

- Through: write to a topic and get a stream / table from the topic

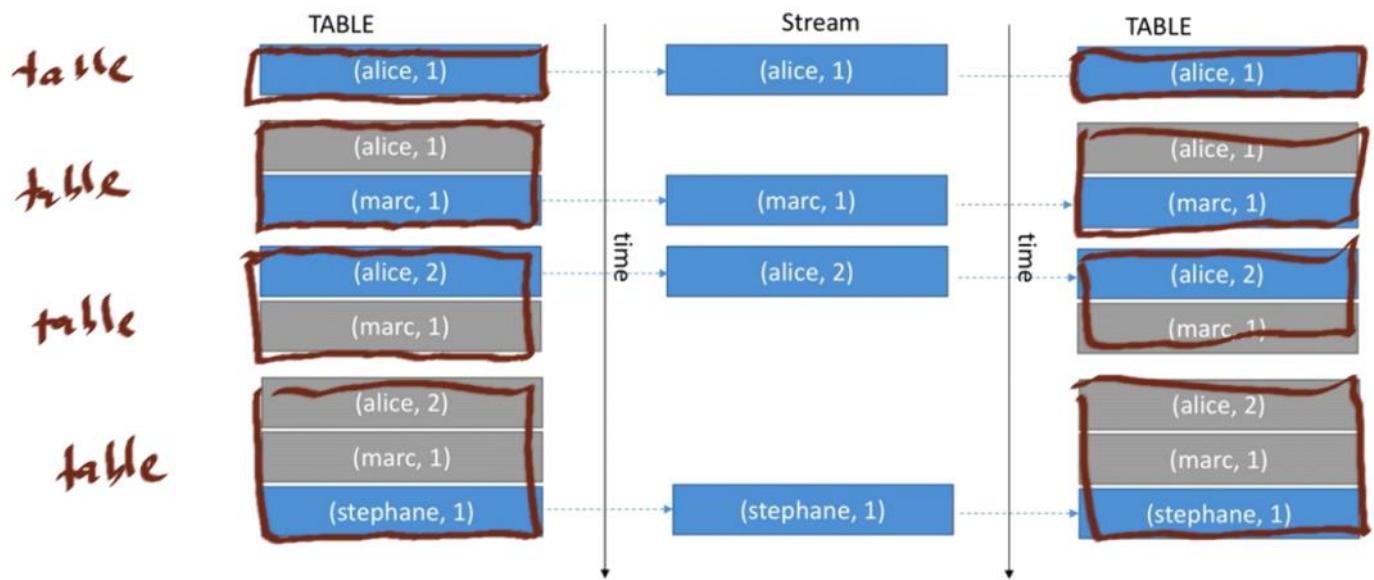
```
KStream<String, Long> newStream = stream.through("user-clicks-topic");
KTable<String, Long> newTable = table.through("my-table-output-topic");
```

KStream & KTable Duality (from confluent docs)



- **Stream as Table:** A stream can be considered a changelog of a table, where each data record in the stream captures a state change of the table.
- **Table as Stream:** A table can be considered a snapshot, at a point in time, of the latest value for each key in a stream (a stream's data records are key-value pairs).

table → stream → table



Transforming a KTable to a KStream



- It is sometimes helpful to transform a KTable to a Kstream in order to keep a changelog of all the changes to the Ktable (see last lecture on Kstream / Ktable duality)
- This can be easily achieved in one line of code!

```
KTable<byte[], String> table = ...;

// Also, a variant of 'toStream' exists that allows you
// to select a new key for the resulting stream.
KStream<byte[], String> stream = table.toStream();
```

Transforming a KStream to a KTable

- Two ways:

- Chain a `groupByKey()` and an aggregation step (`count`, `aggregate`, `reduce`)

```
KTable<String, Long> table = usersAndColours.groupByKey()
    .count();
```

- Write back to Kafka and read as KTable

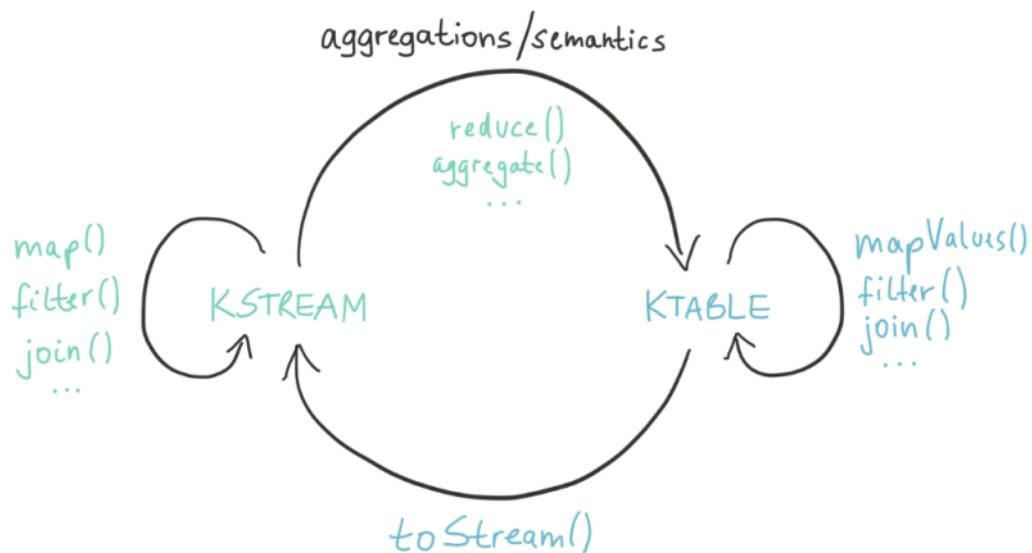
```
// write to Kafka
stream.to("intermediary-topic");

// read from Kafka as a table
KTable<String, String> table = builder.table("intermediary-topic");
```

топология одновременно использует все: stateful+stateless
операции а также и стримы и топики

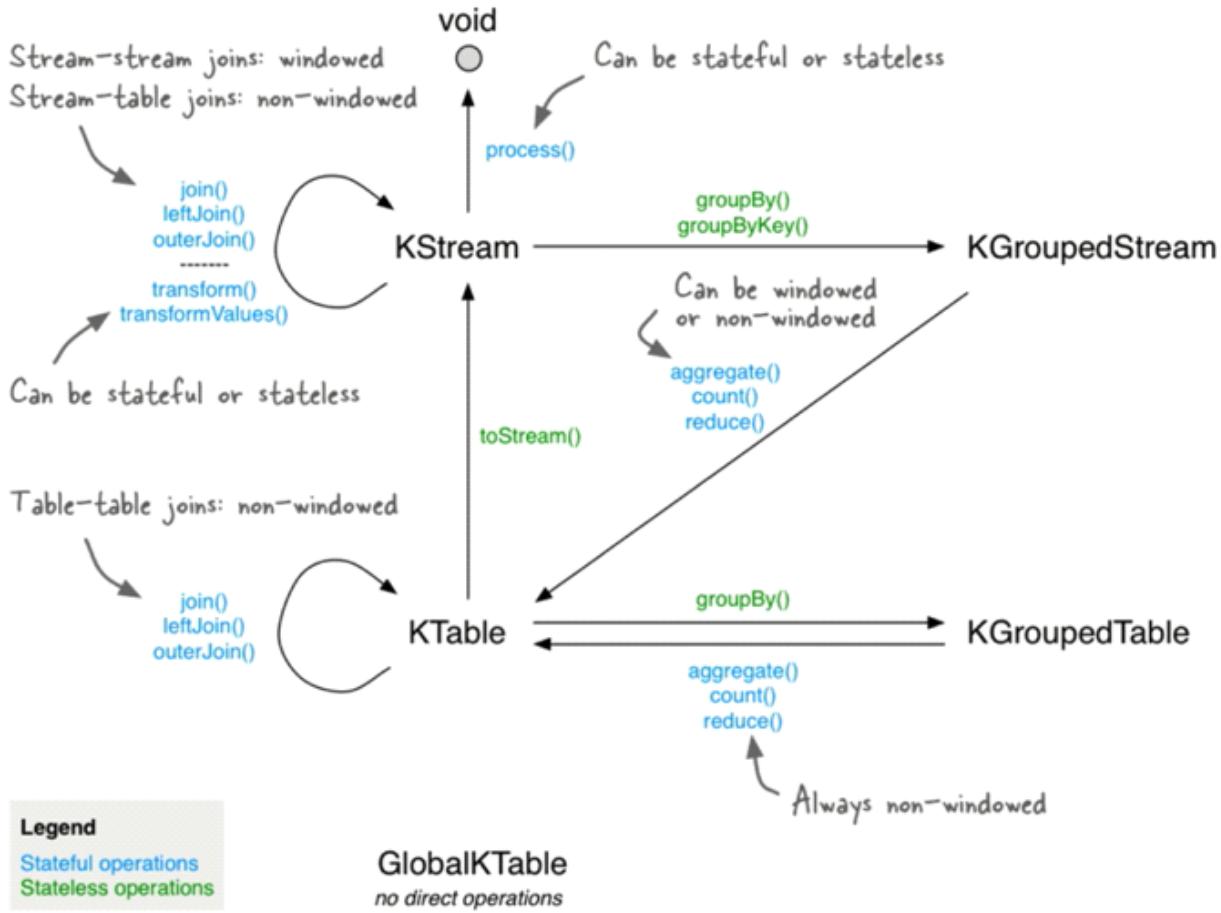
Mixed Processing

- You can build stream processing topologies with mixed stateless and stateful processing



It is important to familiarize yourself with the output types of the various stream processors so that your code can appropriately handle the data.

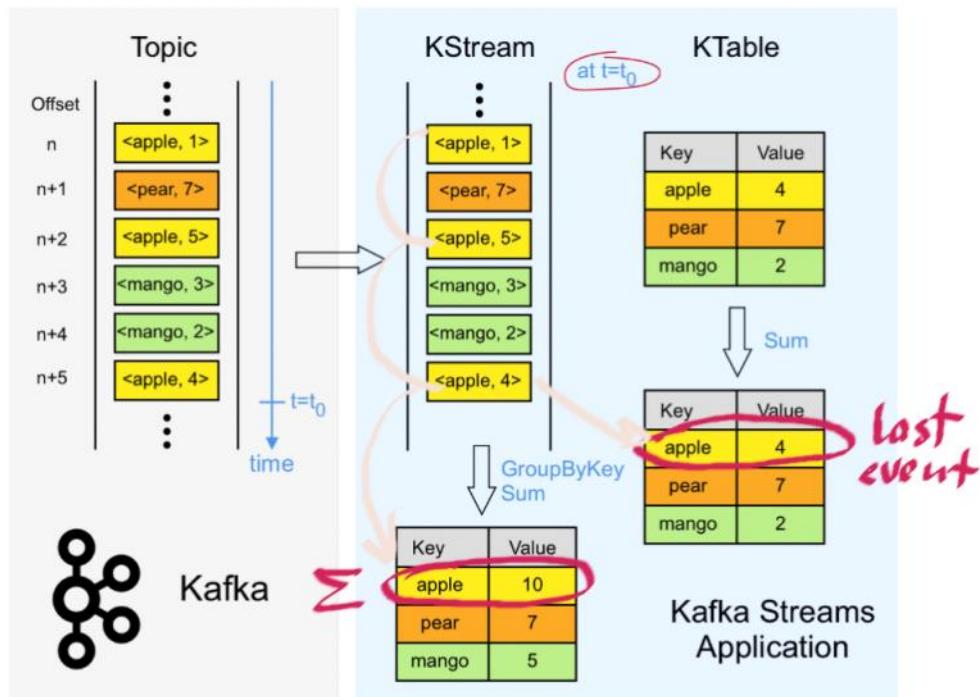
- **join**: used to join two **KStream** objects, a **KStream** and a **KTable**, or two **KTable** objects. This is often used to enrich data coming from one stream with the data of another stream or table (such as a lookup table)
- **aggregate, reduce**: the two functions are similar in what they're doing. We can apply any aggregation function such as summing, averaging, counting, etc. to a group of records. The group is by key and time window (if time windows are used)



пример различий между stream и table

- **events stream** - так как в стриме мы сохраняем все события, то мы например можем посчитать сумму желтых событий (те операции агрегации)
 - доступна вся история всех данных
- **changelog stream** - так в таблице мы перезатираем все предыдущие события новым, то желтое будет содержать только число в последнем желтом событии
 - доступны только последние значения

KStreams and KTables



- A **KStream** is an abstraction of a record stream
 - Each record represents a self-contained piece of data in the unbounded data set
- A **KTable** is an abstraction of a changelog stream
 - Each record represents an update
- Example: We send two records to the stream
 - ('apple', 1), and ('apple', 5)
- If we were to treat the stream as a **KStream** and sum up the values for **apple**, the result would be **6**
- If we were to treat the stream as a **KTable** and sum up the values for **apple**, the result would be **5**
 - The second record is treated as an update to the first, because they have the same key



It is clear that summing a table (as indicated at the center right of the slide) is not possible. It is only a conceptual comparison between a **KStream** and a **KTable**.

таблица - это "медленный" стрим

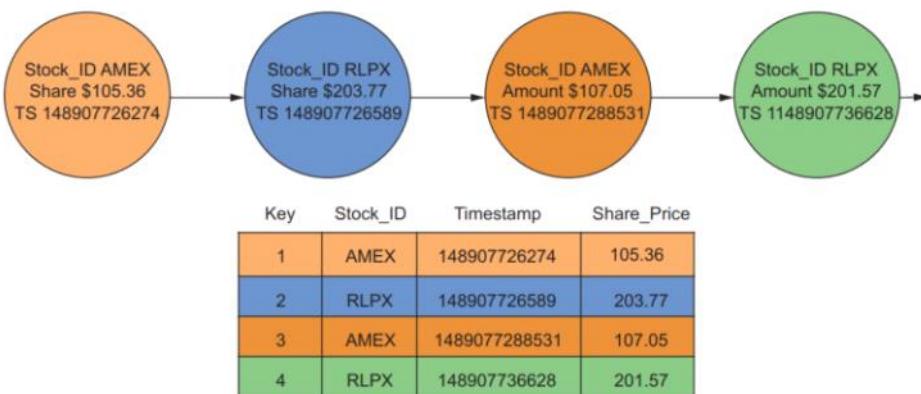
На первый взгляд, сохранение состояния и потоковая обработка между собой не согласуются. Потоковая обработка означает постоянный поток отдельных событий, особо не связанных друг с другом, которые нужно обрабатывать по мере их возникновения. Понятие состояния же скорее вызывает представление о статическом ресурсе вроде таблицы базы данных. На деле же их можно рассматривать как одно и то же. Но изменения в потоке данных происходят намного быстрее и чаще, чем в таблице базы данных!

Event streams vs. update streams

- KStream рассматривает каждую запись как отдельную сущность. KTable же вывел только три записи, поскольку рассматривает их как обновления предыдущих.
- Главный вывод: записи с одинаковыми ключами в потоке данных по своей сути обновления, а не самостоятельные новые записи. Именно понятие потока обновлений лежит в основе интерфейса KTable

Если поток событий мы сравнивали с журналом (**log**), то поток обновлений(те **changelog** это как бы **table**) можно сравнить с журналом **изменений**

- В журнале видны все записи, а в журнале изменений — только последняя запись для каждого ключа
- Различие в том, что журнал используется, когда нужно видеть все записи, а журнал изменений — только когда нужно видеть последнюю запись для каждого ключа.

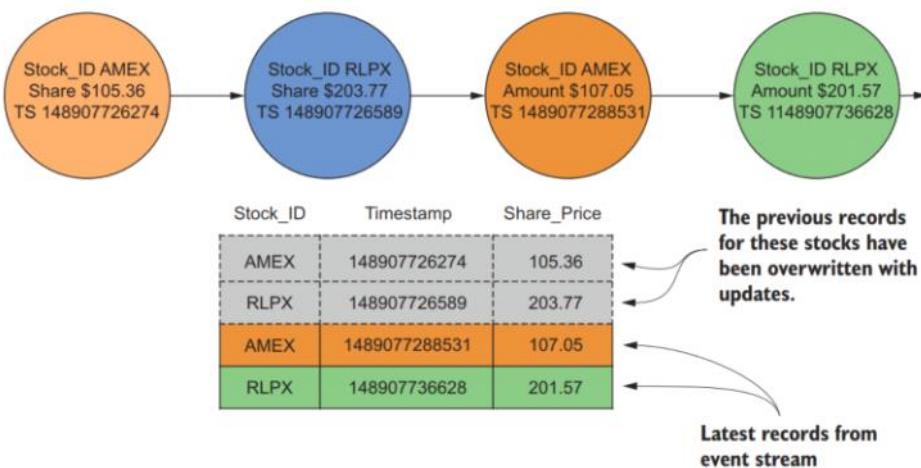


This shows the relationship between events and inserts into a database. Even though it's stock prices for two companies, it counts as four events because we're considering each item on the stream as a singular event.

As a result, each event is an insert, and we increment the key by one for each insert into the table.

With that in mind, each event is a new, independent record or insert into a database table.

Figure 5.3 A stream of individual events compares to inserts into a database table. You could similarly imagine streaming each row from the table.



If you use the stock ID as a primary key, subsequent events with the same key are updates in a changelog. In this case, you only have two records, one per company. Although more records can arrive for the same companies, the records won't accumulate.

Figure 5.4 In a changelog, each incoming record overwrites the previous one with the same key. With a record stream, you'd have a total of four events, but in the case of updates or a changelog, you have only two.

KTable использует для хранения локальное состояние в сочетании с Kafka Streams

```
builder.table(STOCK_TICKER_TABLE_TOPIC);
```

С помощью этого простого оператора `StreamsBuilder.table` создает экземпляр `KTable` и одновременно незаметно создает объект `StateStore` для отслеживания состояния потока, таким образом создавая поток обновлений. У созданного при этом подходе хранилища состояния `StateStore`, недоступного для интерактивных запросов, есть только внутреннее имя.

Существует перегруженная версия метода `StreamsBuilder.table`, принимающая в качестве параметра экземпляр `Materialized`, благодаря чему у вас появляется возможность настроить тип хранилища и задать для него имя, чтобы сделать его доступным для запросов. Мы обсудим интерактивные запросы позднее в этой главе.

кеш таблиц

Кэширование KTable удаляет избыточные обновления записей с одинаковым ключом, предотвращая таким образом переполнение дочерних узлов KTable в топологии непрерывными обновлениями

- в хранилище состояния помещаются только самые свежие обновления, что означает весьма существенный рост производительности при использовании постоянных хранилищ состояния
- Кэш объекта KTable служит для дедупликации обновлений записей с одним ключом. Благодаря этой дедупликации дочерние узлы получают только самые свежие обновления вместо всех обновлений, что существенно снижает объемы обрабатываемых данных.
- Поскольку KTable соответствует журналу изменений в потоке данных, можно ожидать, что в каждый конкретный момент придется иметь дело только с последним изменением. А применение кэша обеспечивает реализацию этого поведения. При необходимости обработать все записи в потоке лучше воспользоваться описанным ранее интерфейсом KStream.

Incoming stock ticker record

YERB	105.36
------	--------

As records come in, they are also placed in the cache, with new records replacing older ones.

Cache

YERB	105.24
NDLE	33.56
YERB	105.36

Figure 5.7 KTable caching deduplicates updates to records with the same key, preventing a flood of consecutive updates to child nodes of the KTable in the topology.

Отправка записей дальше по конвейеру происходит в результате или фиксации, или достижения максимального размера кэша

- Как видите, необходим компромисс между размером кэша и интервалом фиксации.
- Большой размер кэша при коротком интервале фиксации приведет к частым обновлениям.
- А более длинный интервал фиксации может привести к меньшему числу обновлений (в зависимости от настроек памяти), поскольку для освобождения памяти предусмотрен механизм вытеснения кэша.
- Жестких правил тут не существует — узнать, что лучше подходит для вас, можно только методом проб и ошибок. Рекомендую начать со значений по умолчанию: 30 секунд (интервал фиксации) и 10 Мбайт (размер кэша).

(a) Размер кэша определяется настройкой `cache.max.bytes.buffering`

- Чем больше кэш, тем меньше изменений будет отправляться далее. Кроме того, кэширование снижает объем данных, записываемых на диск постоянными хранилищами (RocksDB), а в случае включенного журналирования — и число записей для любого хранилища, отправляемых в топик журнала изменений.
- Размер кэша определяется настройкой `cache.max.bytes.buffering`, которая задает объем памяти, выделяемой под кэш записей. Заданное количество памяти поровну распределяется между потоками выполнения для этого потока данных (число потоков выполнения для потока данных задается параметром `StreamsConfig.NUM_STREAM_THREADS_CONFIG` конфигурации, равным по умолчанию 1).
- Для отключения кэширования необходимо установить параметр `cache.max.bytes.buffering` в 0. Но в результате этого далее по конвейеру будут отправляться все обновления KTable, то есть поток журнала изменений фактически

превратится в поток событий. Кроме того, без кэширования будет больше операций ввода/вывода, ведь постоянным хранилищам придется записывать на диск все обновления, а не только последние.

- Отправка записей дальше по конвейеру происходит в результате или фиксации, или достижения максимального размера кэша. И наоборот, отключение кэширования приведет к отправке дальше по конвейеру всех записей, включая записи с дублирующимися ключами. Вообще говоря, при использовании KTable лучше, чтобы кэширование было включено.

(b) `commit.interval.ms` - частоту (в миллисекундах) сохранения состояния узла-обработчика

- При сохранении (фиксации) состояния узла-обработчика происходит сброс кэша на диск с отправкой дальше по конвейеру последних обновленных и дедуплицированных записей.

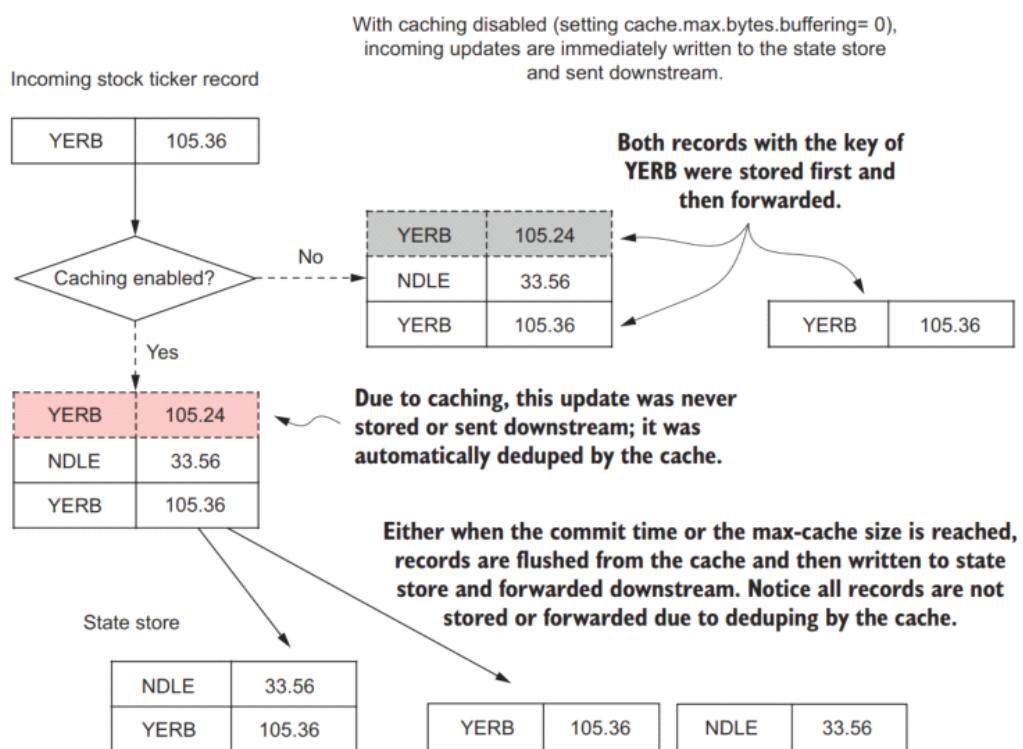


Figure 5.8 Full caching workflow: if caching is enabled, records are deduped and sent downstream on cache flush or commit.

При отключенном кэшировании (параметр cache.max.bytes.buffering=0)
поступающие обновления сразу же записываются
в хранилище состояния и отправляются дальше по конвейеру

Входящая запись
биржевого тикера

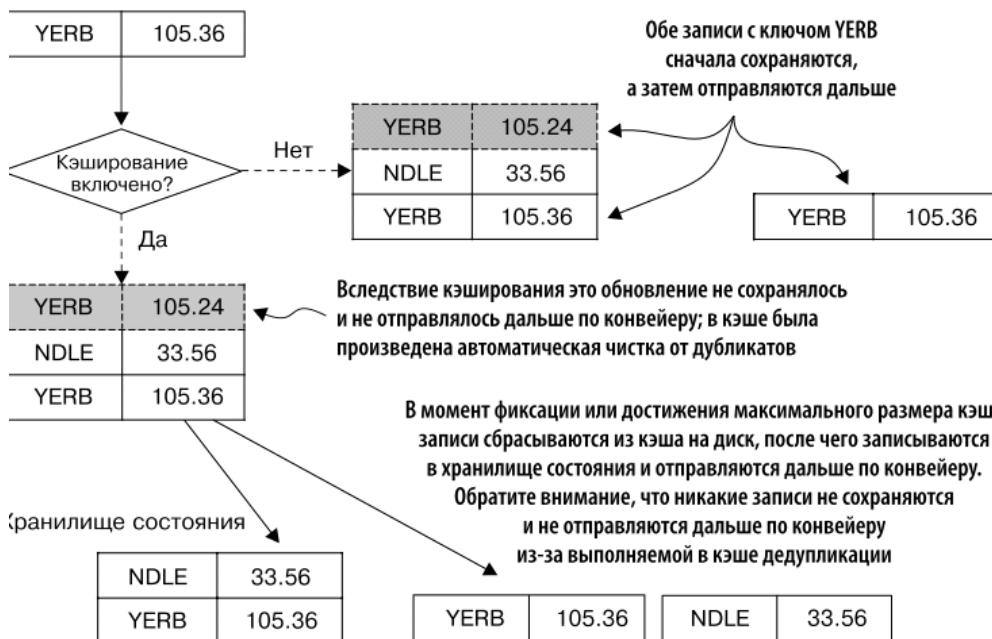


Рис. 5.8. Полный технологический процесс кэширования: если кэширование включено, записи с дублированием удаляются и отправляются дальше по конвейеру при сбросе кэша на диск или фиксации

заметки из книги

Streams → tables

The aggregation of a stream of updates over time yields a table.

Tables → streams

The observation of changes to a table over time yields a stream.

Таблицы - это данные в состоянии покоя .

Потоки - это данные в движении .

- таблицы отражают представление набора данных в целом в определенный момент времени
- потоки подобны математической-производным таблиц, а таблицы - интегралам потоков,

- Tables are data at rest.

This isn't to say tables are static in any way; nearly all useful tables are continuously changing over time in some way. But at any given time, a

snapshot of the table provides some sort of picture of the dataset contained together as a whole.² In that way, tables act as a conceptual resting place for data to accumulate and be observed over time. Hence, data at rest.

- Streams are data *in motion*.

Whereas tables capture a view of the dataset as a whole at a *specific point in time*, streams capture the evolution of that data *over time*. Julian Hyde is fond of saying streams are like the derivatives of tables, and tables the integrals of streams, which is a nice way of thinking about it for you math-minded individuals out there. Regardless, the important feature of streams is that they capture the inherent movement of data within a table as it changes. Hence, data in motion.

общая теория относительности таблиц и потоков

A general theory of stream and table relativity:

- Data processing pipelines (both batch and streaming) consist of *tables*, *streams*, and *operations* upon those tables and streams.
- Tables are *data at rest*, and act as a container for data to accumulate and be observed over time.
- Streams are *data in motion*, and encode a discretized view of the evolution of a table over time.
- Operations act upon a stream or table and yield a new stream or table. They are categorized as follows:

- stream → stream: Nongrouping (element-wise) operations

Applying *nongrouping* operations to a stream alters the data in the stream while leaving them in motion, yielding a new stream with possibly different cardinality.

- stream → table: Grouping operations

Grouping data within a stream brings those data to rest, yielding a *table* that evolves over time.

- *Windowing* incorporates the dimension of event time into such groupings.

◦ *Merging windows* dynamically combine over time, allowing them to reshape themselves in response to the data observed and dictating that key remain the unit of atomicity/parallelization, with window being a child component of grouping within that key.

- table → stream: Ungrouping (triggering) operations

Triggering data within a table ungroups them into motion, yielding a *stream* that captures a view of the table's evolution over time.

- *Watermarks* provide a notion of input completeness relative to event time, which is a useful reference point when triggering event-timestamped data, particularly data grouped into event-time windows from unbounded streams.

◦ The *accumulation mode* for the trigger determines the nature of the stream, dictating whether it contains deltas or values, and whether retractions for previous deltas/values are provided.

- table → table: (none)

There are no operations that consume a table and yield a table, because it's not possible for data to go from rest and back to rest without being put into motion. As a result, all modifications to a table are via conversion to a stream and back again.

окна - это всего лишь группировка

- windowing is a modification of grouping by key, in which the window becomes a secondary part of a hierarchical key

- Что мне нравится в этих правилах, так это то, что они имеют смысл. У них очень естественное и интуитивное ощущение, и в результате они значительно упрощают понимание того, как данные передаются (или нет) через последовательность операций.
- Они кодифицируют тот факт, что **данные существуют в одной из двух структур в любой момент времени (потоки или таблицы)**, и

- предоставляют простые правила для рассуждений о переходах между этими состояниями.
- Они демистифицируют окна, показывая, что это всего лишь небольшая модификация вещи, которую все уже понимают от рождения: **группировка**. Они подчеркивают, почему операции группировки в целом всегда являются камнем преткновения для потоковой передачи (потому что они переносят данные в потоки для хранения в виде таблиц), но также очень ясно дают понять, какие виды операций необходимы, чтобы что-то развалилось (триггеры; т. Е. Операции **разгруппировки**).
- Классифицируя операции таким образом, становится тривиальным понять, как данные проходят через (и задерживаются внутри) данного конвейера с течением времени.

stream → stream

Nongrouping (element-wise) operations

stream → table

Grouping operations

table → stream

Ungrouping (triggering) operations

table → table

(nonexistent)

вы можете довольно легко переместить данные в более простые окна в SQL, поскольку они существуют сейчас, просто включив время как часть параметра GROUP BY

- Как мы узнали из главы 6 , работа с окнами - это модификация группировки по ключу, при которой окно становится второстепенной частью иерархического ключа.
- Как и в случае с классической программной пакетной обработкой, вы можете довольно легко переместить данные в более простые окна в SQL, поскольку они существуют сейчас, просто включив время как часть параметра GROUP BY .
- если мы можем неявно использовать оконные конструкции, используя существующие конструкции SQL, зачем вообще беспокоиться о поддержке явных оконных конструкций?
 - Окно берет на себя математические вычисления окон за вас. Намного легче последовательно делать вещи правильно, если вы напрямую указываете основные параметры, такие как ширина и слайд, чем самостоятельно вычисляете математику окна. 14
 - Использование окон позволяет кратко выразить более сложные динамические группировки, такие как сеансы. Несмотря на то, что SQL технически способен выражать отношения каждый элемент в пределах некоторого временного промежутка другого элемента, которые определяют окна сеанса, соответствующее заклинание представляет собой запутанный беспорядок аналитических функций, самосоединений и распаковки массива, который ни один простой смертный не мог разумно ожидать, что будет колдовать самостоятельно.

Example 8-2. Summation pipeline

```
PCollection<String> raw = IO.read(...);
PCollection<KV<Team, Integer>> input = raw.apply(new ParseFn());
PCollection<KV<Team, Integer>> totals = input
    .apply(Window.into(FixedWindows.of(TWO_MINUTES)))
    .apply(Sum.integersPerKey());
```



Чтобы сделать то же самое в SQL, у нас есть два варианта: неявно окно, включив некоторую уникальную функцию окна (например, временную метку окончания) в оператор GROUP BY , или использовать встроенную операцию управления окнами.

операции группирования всегда потребляют поток и дают таблицу.

- объединяя данные, которые имеют общее свойство (например, ключ), мы собираем вместе некоторое количество ранее не связанных отдельных элементов данных в группу связанных элементов.
- И, как мы узнали из главы 6 , операции группирования всегда потребляют поток и дают таблицу.
- Зная эти две вещи, можно сделать лишь небольшой шаг, чтобы прийти к выводу, который составляет основу всей этой главы: по сути, все объединения являются потоковыми объединениями .
- Что замечательно в этом факте, так это то, что он фактически делает тему потоковых подключений гораздо более доступной. Все инструменты, которые мы изучили для рассуждений о времени в контексте операций группировки потоковой передачи (создание окон, водяные знаки, триггеры и т. д.), Продолжают применяться в случае потоковых соединений. Что, возможно, пугает, так это то, что добавление потоковой передачи в микс, похоже, может только усложнить ситуацию. Но, как вы увидите в следующих примерах, моделирование всех объединений как потоковых объединений отличается элегантной простотой и согласованностью
- Вместо того чтобы думать, что существует множество различных подходов к объединению, становится ясно, что почти все типы объединений на самом деле сводятся к незначительные вариации по той же схеме

global table

23 декабря 2020 г. 14:51

global ktables рекомендуется использовать, пока они менее 1 ГБ
(тк они не масштабируются)

- к обычным ktables ограничение в 1 ГБ тоже относится, лучше сделать столько партиций чтобы в итоге на каждой streams-проги local state store занимал не более нескольких гигабайт
- На самом деле в Kafka Streams есть два типа таблиц: KTables и Global KTables. Когда запущен только один экземпляр службы, они фактически ведут себя одинаково. Однако, если мы масштабируем нашу службу - так, чтобы у нее было, скажем, четыре параллельных экземпляра, - мы бы увидели немного другое поведение. Это связано с тем, что **клонируются глобальные таблицы KTables: каждый экземпляр микросервиса получает полную копию всей таблицы.** Обычные KTables сегментированы: набор данных распространяется по всем экземплярам службы. Короче говоря, **Global KTables проще в использовании, но у них есть ограничения масштабируемости**, поскольку они **клонируются между машинами**, поэтому используйте их для таблиц поиска (**обычно до нескольких гигабайт**), которые легко поместятся на локальном диске машины. Используйте KTables и масштабируйте свои сервисы, когда набор данных больше.
- На самом деле в Kafka Streams есть два типа таблиц: KTables и Global KTableles. Когда запущен только один экземпляр службы, они ведут себя одинаково. Однако, если мы масштабируем нашу службу - так, чтобы у нее было четыре экземпляра, работающих параллельно, - мы бы увидели несколько иное поведение. Это связано с тем, что **глобальные таблицы KTables broadcasted**: каждый экземпляр службы получает полную копию всей таблицы. Обычные KTables разделены: набор данных распространяется по всем экземплярам службы.

на каждом клиенте **будет полная** копия глобальной таблицы
а обычные Ktable просто шардируются ?поパーティциям? те можно смасштабировать
обычных репартиционированием

broadcasted table vs partitioned table

- так как global table есть на каждой ноде, то мы ее можем соединять с любой таблицей/потоком
- для соединения обычных ktable может потребоваться **repartition(by key)** одного из потоков
- То, является ли таблица широковещательной или секционированной, влияет на то, как она может выполнять соединения.
- С помощью Global KTable, поскольку вся таблица существует на каждом узле, мы можем присоединиться к любому атрибуту, который пожелаем, подобно присоединению по внешнему ключу в базе данных.
- Это не так в KTable. Поскольку он разбит на разделы, к нему можно присоединиться

только по его первичному ключу, точно так же, как вы должны использовать первичный ключ при объединении двух потоков. Итак, чтобы присоединиться к KTable или потоку по атрибуту, который не является его первичным ключом, мы должны выполнить repartition.

- So, in short, Global KTables work well as lookup tables or star joins but take up more space on disk because they are broadcast. KTables let you scale your services out when the dataset is larger, but may require that data be rekeyed.¹

distributed processing and fault tolerance

19 декабря 2020 г. 11:21

What Is the Kafka Streams API?

- **Transforms and enriches data**
 - per-record stream processing
 - millisecond latency
 - stateless & stateful processing
 - windowing operations
- **Fault-tolerant and distributed processing**
- Has **Domain-Specific Language (DSL)**
 - High level operations: `map`, `flatMap`, `count`, etc.
- **Processor API** for even more flexibility

This module is intended as a very basic overview of Kafka Streams and KSQL due to time constraints. More advanced topics (e.g., Global KTables, User Defined Functions (UDFs), **STRUCT data type**) are beyond the scope of this course and are discussed in the *Stream Processing with Kafka Streams and KSQL* course.

Additional information:

- In Kafka Streams there's also a Processor API in addition to the DSL providing even more flexibility.
- **Windowing operations are not the equivalent of micro batching found in other streaming frameworks**
 - The term "micro-batch" is frequently used to describe scenarios where batches are small and/or processed at small intervals. Even though processing may happen as often as once every few minutes, data is still processed a batch at a time. Spark Streaming is an example of a system that supports micro-batch processing.
 - **Stream processing with windowing** in turn does not wait until the end of a time window until results are published. Results are continuously updated (e.g. aggregated) and published for downstream processing.



тк стримы-это просто джава -прога (как консьюмер берет из кафки и как продьюсер кладет обратно в кафку)

- масштабирование достигается за счет увеличения числа прог(консьюмеров)(можно в кубере просто увеличить число реплик)
- **fault-tolerance** достигается за счет обычной перегруппировки консьюмеров, если один из них отвалится

A Library, Not a Framework

- Kafka Streams provides **streaming capabilities**
- Other streaming frameworks:
 - Apache Flink
 - Spark Streaming
 - Apache Storm
 - Apache Samza
 - etc.
- **BUT:** Kafka Streams does not require its own cluster
 - **No need for Zookeeper**

A Library, Not a Framework

- Kafka Streams provides **streaming capabilities**
- Other streaming frameworks:
 - Apache Flink
 - Spark Streaming
 - Apache Storm
 - Apache Samza
 - etc.
- **BUT:** Kafka Streams does not require its own cluster
 - It's just a **Java library**
 - Runs on **1...n machines**
- Using Kafka Streams may reduce load on remote DBMS
 - Local data accesses via the local persistent datastore



What is meant with "Kafka Streams does not require its own cluster"? It is clear that we always need a cluster of brokers, no matter whether we use Kafka Streams or say Flink for stream processing. The meaning is that Flink needs its own "Flink Cluster infrastructure" whilst a Kafka Streams application is just a Java application and thus can run without extra clustering infrastructure. Of course one can use e.g. Kubernetes to run the streams app at scale.

Regarding the reduction on load on a remote DBMS:

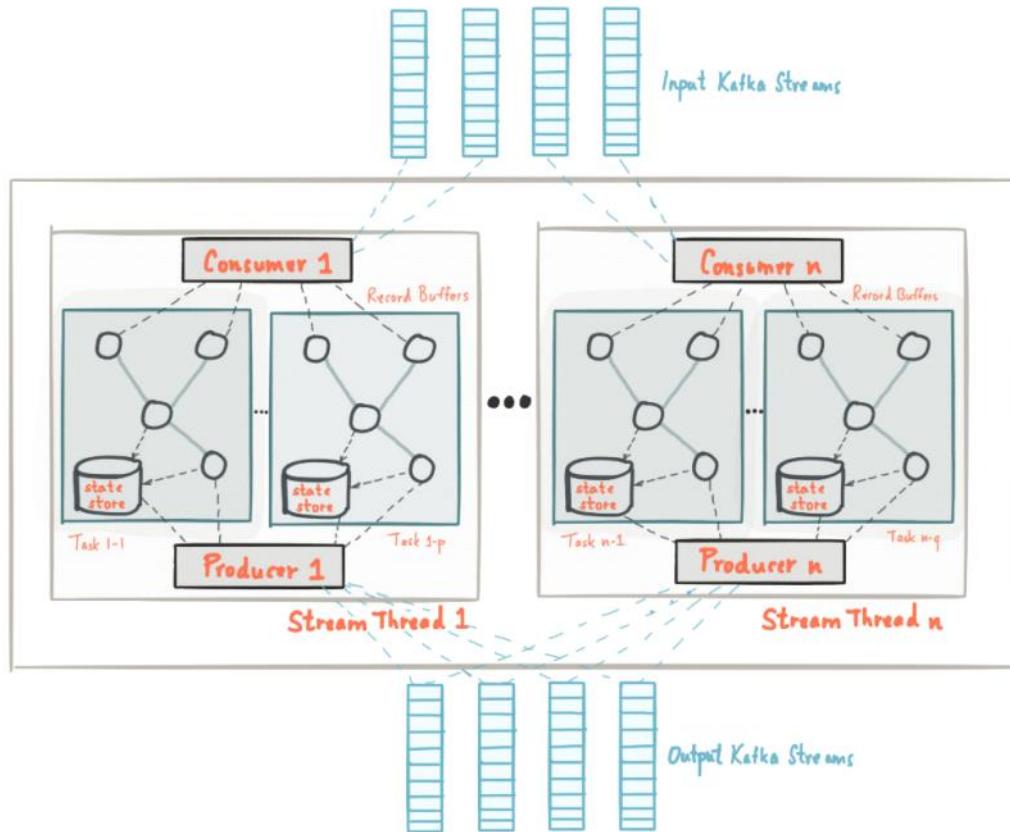
- **Old way:** processor looks up reference data in a remote database management system (e.g. Cassandra) or even uses that remote system to merge joined results.
- **New way:** Migrate some of the processing jobs to Kafka Streams API to reduce the load on the remote database management system by turning remote data accesses into cheaper local data accesses.

Why Not Just Build Your Own?

- Many people are currently building their own stream processing applications
 - Using the Producer and Consumer APIs
- Using Kafka Streams API is much easier than taking the 'do it yourself' approach
 - Well-designed, well-tested, robust
 - Means you can focus on the application logic, not the low-level plumbing

Kafka Streams is part of the open source Apache Kafka distribution and so is available for free as part of that package.

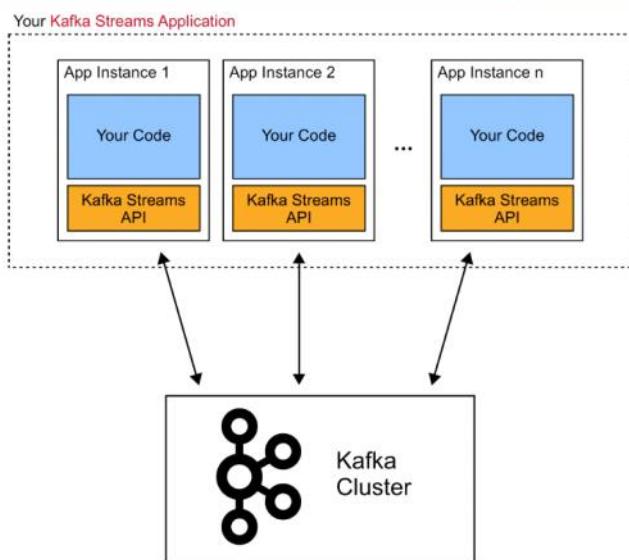
Logical View of Kafka Streams Application



This diagram is a logical view of a Kafka Streams application that contains multiple stream threads, each of which in turn containing multiple stream tasks. Stream threads can run in parallel on a single system or distributed across multiple systems for performance and availability. Stateful information is stored locally in a RocksDB temporarily before being persisted to a special Kafka topic.

- vertical scaling `num.stream.threads=1`: можем увеличить количество тредов в стримс-проге
- horizontal scaling: а можем увеличить количество прог (у каждой проги по дефолту по одному треду)

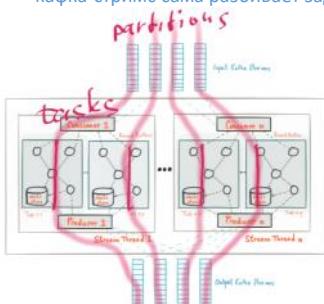
Kafka Streams Application Architecture



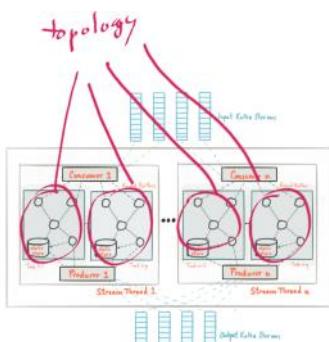
- The Streams API of Apache Kafka, available through a Java library, can be used to build highly scalable, elastic, fault-tolerant, distributed applications and microservices.
- A unique feature of the Kafka Streams API is that the applications we build with the Kafka Streams library are normal Java applications. These applications can be packaged, deployed, and monitored like any other Java application – there is no need to install separate processing clusters or similar special-purpose and expensive infrastructure!
- We can run more than one instance of the application. The instances run independently but will automatically discover each other and collaborate.
- We can elastically add or remove instances during live operation.
- If one instance (unexpectedly dies) the others will automatically take over its work and continue where it left off.

task - это единица параллелизма и масштабирования в стрим-апи

- одна таска на одну партицию топика
- те в максималке: **стримс-прог столько же сколько партиций**
- в идеале на один thread должна быть одна таска
- если количество стримс-прог меньше чем количество партиций, тогда в одну стримс-прогу могут попасть несколько тасков
- кафка-стримс сама разбивает задачу на таски в зависимости от количества партиций входного топика



- каждая таска содержит топологию целиком (те каждая таска делает одинаковую обработку данных)



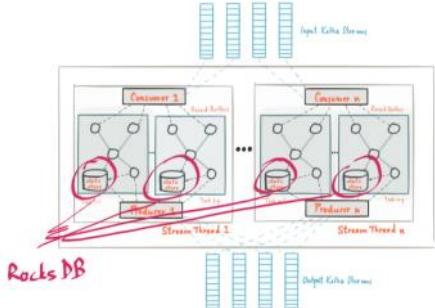
local state store - нужно для

- сохранения локального состояния при вычислениях (durability),
- также для кеша,
- для операций агрегации и других **stateful** операций

эта БД локальная - те находится на том же сервере что и стрим-прога

rocksDB делает бэкап в кафка changelog compacted-топик, на тот случай если стримс-клиент упадет и удалится целиком (тогда новый инстанс клиента поднимается и делает replaying из changelog topic)

но в кубере лучше сделать persistent_volume в statefulset (чтобы стрим-прога быстрее поднялась)



- Stateful client applications may use Streams operations (e.g., aggregates, joins, windows)
- Clients keep state in local state stores (*i.e.*, RocksDB)
 - The state stores use local disk on the client machines
 - The state stores have 50MB - 100MB memory overhead
 - Clients persist state stores to a compacted Kafka Topic
 - If the client state store fails, the data is recoverable
 - Using standby replicas for Kafka Streams tasks will result in extra copies of state stores
- If client performance is bound by CPU
 - Add cores or machines, and increase the number of threads
 - `num.stream.threads` (Default: 1)
- If client performance is bound by network, memory, or disk
 - Scale out the clients onto additional machines

если стрим-прога упала то таск автоматически поднимется на оставшемся живом сервере (похоже на обычный consumer rebalance)

- You can run Kafka streaming applications across multiple machines
- Parallelizing Kafka Streams or KSQL applications can improve throughput and performance
- Kafka handles dividing the work across the machines
- It also provides fault tolerance
 - If a machine fails, the task it was working on is automatically restarted on one of the remaining instances
- The number of tasks in the Kafka Streams application may be increased by increasing the number of Partitions

Kafka Streams is designed around parallelization of work loads. Systems running Streams applications should be multi-core in order to take best advantage of this design.

`num.standby.replicas=0` standby replica tasks feature, число простояющих тасков "про запас" (они позволяют сверхбыстро подняться новому инстансу)

- Clients can enable standby replica tasks for the streaming applications
 - These standby replica tasks have fully replicated copies of the state
 - When a task migration happens, Kafka Streams attempts to assign a task to an application instance where such a standby replica already exists
 - Reduces restoration time for a failed task
- `num.standby.replicas`: number of standby replicas for each task (Default: 0)

The use of standby replicas is recommended for performance sensitive environments.

?? что таски представляют собой внутри, разве consumer rebalancing недостаточно?

- Если вам нужна большая вычислительная мощность для вашего приложения потоковой обработки, вы можете просто запустить другой экземпляр вашего приложения потоковой обработки, например, на другом компьютере, для масштабирования. Экземпляры вашего приложения узнают друг о друге и автоматически начнут разделять работу по обработке. Более конкретно, то, что будет передано от существующих экземпляров новым экземплярам, - это (некоторые из) потоковые задачи, которые были запущены существующими экземплярами. Перемещение потоковых задач из одного экземпляра в другой приводит к перемещению работы по обработке плюс любое внутреннее состояние этих потоковых задач (состояние потоковой задачи будет воссоздано в целевом экземпляре путем восстановления состояния из соответствующей темы журнала изменений - changelog topic).

windowing, joining, aggregations - основные операции над стримом

19 декабря 2020 г. 14:06

Windowing, Joining, and Aggregations

- Kafka Streams API allows us to *window* the stream of data **by time**
 - To divide it up into 'time buckets'
- We can aggregate records **by key 1 stream**
 - Combine multiple input records together in some way into a single output record
 - Examples: sum, count
 - This is usually done on a windowed basis
- We can join, or *i.e.*, merge, data from different sources **by key 2 streams**

Stream processing with **windowing** - используется в кафке
micro batching - используется в **Spark Streaming**

- Windowing отличается от батчинга тем что как только результаты склеиваются в окне они сразу поступают клиенту (а батчинг ждет окончания обработки порции)

- Kafka Streams provides **streaming capabilities**
- Other streaming frameworks:
 - Apache Flink
 - Spark Streaming
 - Apache Storm
 - Apache Samza
 - etc.

Оконные операции дают возможность собирать данные **порциями**, в отличие от сбора неограниченных данных

- Оконная обработка данных (**windowing**) и обработка по корзинам (**bucketing**) — в чем-то синонимичные понятия. И то и другое относится к разбиению информации на меньшие порции или категории.
- **Оконная обработка данных подразумевает категоризацию по времени**, но результат в обоих случаях одинаков.

`processing.guarantee=exactly_once` EOS включается всего одной настройкой

Kafka Streams Processing Guarantees

- Supports **at-least-once** processing
- With no failures, it will process data exactly once
- Upon failure, some records may be processed **more than once**
- Supports **Exactly Once** processing semantics (EOS)
 - Set `processing.guarantee=exactly_once` (Default: `at_least_once`)

Whether or not it is acceptable to process certain records **more than once** depends on the use-case!



It is worth mentioning the **robustness** of exactly once processing (EOS) in KStreams vs plain consumer groups!



WordCount Internal Topics

- Running a Kafka Streams may eventually create internal intermediary topics.
- Two types:
 - Repartitioning topics: in case you start transforming the key of your stream, a repartitioning will happen at some processor.
 - Changelog topics: in case you perform aggregations, Kafka Streams will save compacted data in these topics
- Internal topics:
 - Are managed by Kafka Streams
 - Are used by Kafka Streams to save / restore state and re-partition data
 - Are prefixed by application.id parameter
 - **Should never be deleted, altered or published to. They are internal**

```
~/kafka_2.12-0.11.0.0> bin/kafka-topics.sh --list --zookeeper localhost:2181
__consumer_offsets
streams-file-input
streams-wordcount-Counts-changelog
streams-wordcount-Counts-repartition
streams-wordcount-output
word-count-input
word-count-output
wordcount-application-Counts-changelog
wordcount-application-Counts-repartition
```

?? в какой момент создаются внутренние топики

```
KStreamBuilder builder = new KStreamBuilder();
// 1 - stream from Kafka

KStream<String, String> textLines = builder.stream("word-count-input");
KTable<String, Long> wordCounts = textLines
    // 2 - map values to Lowercase
    .mapValues(textLine -> textLine.toLowerCase())
    // can be alternatively written as:
    // .mapValues(String::toLowerCase)
    // 3 - flatmap values split by space
    .flatMapValues(textLine -> Arrays.asList(textLine.split("\\W+")))
    // 4 - select key to apply a key (we discard the old key)
    .selectKey((key, word) -> word)
    // 5 - group by key before aggregation
    .groupByKey()
    // 6 - count occurrences
    .count("Counts");

// 7 - to in order to write the results back to kafka
wordCounts.to(Serdes.String(), Serdes.Long(), "word-count-output");

KafkaStreams streams = new KafkaStreams(builder, config);
streams.start();
```

input topic

output topic

WordCount Streams App Topology



- Let's write the topology using the High Level DSL for our application
- Remember data in Kafka Streams is < **Key , Value** >

1. Stream from Kafka < null, "Kafka Streams">
2. MapValues lowercase < null, "kafka streams" >
3. FlatMapValues split by space < null, "kafka", <null, "kafka", <null, "streams">
4. SelectKey to apply a key < "kafka", "kafka", <"kafka", "kafka", <"streams", "streams">
5. GroupByKey before aggregation (< "kafka", "kafka", <"kafka", "kafka", <"streams", "streams">)
6. Count occurrences in each group < "kafka", 2 >, <"streams", 1>
7. To in order to write the results back to Kafka data point is written to Kafka

```
~/kafka_2.12-0.11.0.0> bin/kafka-console-producer.sh --broker-list localhost:9092 --topic word-count-input
>hello kafka streams
>kafka streams is working
>stephane is kafka level awesome
>
```

```
[x ~] ~/kafka_2.12-0.11.0.0> bin/kafka-console-consumer.sh --bootstrap-server localhost:9092 \
--topic word-count-output \
--from-beginning \
--formatter kafka.tools.DefaultMessageFormatter \
--property print.key=true \
--property print.value=true \
--property key.deserializer=org.apache.kafka.common.serialization.StringDeserializer \
--property value.deserializer=org.apache.kafka.common.serialization.LongDeserializer
streams 2
stephane      1
is            2
kafka         3
level         1
awesome       1
hello         1
working[1
working 4
```

java app 1

30 ноября 2020 г. 20:10

Secure | <https://mvnrepository.com/artifact/org.apache.kafka/kafka-streams>

REPOSITORY

Search for groups, artifacts, categories

Artifacts (11.5M)

Home > org.apache.kafka > kafka-streams

Apache Kafka
Apache Kafka

License: Apache 2.0
Tags: kafka, streaming, apache
Used By: 96 artifacts

Central (17) Hortonworks (784)

Version	Repository	Usages	Date
2.0.x 2.0.0	Central	10	Jul, 2018
1.1.1	Central	8	Jul, 2018
1.1.x 1.1.0	Central	23	Mar, 2018

Categories

oriented
networks
in Metrics
is
Libraries
Line Parsers
implementations

f

The screenshot shows the IntelliJ IDEA interface with the project 'streams-starter-project' open. The left sidebar displays the project structure, including 'streams-starter-project' (containing .idea, src, main, test, and pom.xml), and 'External Libraries'. The right pane shows the contents of the 'pom.xml' file.

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <groupId>com.github.simplesteph.udemy.kafka.streams</groupId>
    <artifactId>streams-starter-project</artifactId>
    <version>1.0-SNAPSHOT</version>
    <dependencies>
        <!-- https://mvnrepository.com/artifact/org.apache.kafka/kafka-streams -->
        <dependency>
            <groupId>org.apache.kafka</groupId>
            <artifactId>kafka-streams</artifactId>
            <version>0.11.0.0</version>
        </dependency>
        <!-- https://mvnrepository.com/artifact/org.slf4j/slf4j-api -->
        <dependency>
            <groupId>org.slf4j</groupId>
            <artifactId>slf4j-api</artifactId>
            <version>1.7.25</version>
        </dependency>
        <!-- https://mvnrepository.com/artifact/org.slf4j/slf4j-log4j12 -->
        <dependency>
            <groupId>org.slf4j</groupId>
            <artifactId>slf4j-log4j12</artifactId>
            <version>1.7.25</version>
        </dependency>
    </dependencies>
</project>
```

```
package com.github.simplesteph.kafka.tutorial4;
```

```
import com.google.gson.JsonParser;
import org.apache.kafka.common.protocol.types.Field;
import org.apache.kafka.common.serialization.Serdes;
import org.apache.kafka.streams.KafkaStreams;
import org.apache.kafka.streams.StreamsBuilder;
import org.apache.kafka.streams.StreamsConfig;
import org.apache.kafka.streams.KStream;
```

```

import java.util.Properties;

public class StreamsFilterTweets {

    public static void main(String[] args) {
        // create properties
        Properties properties = new Properties();
        properties.setProperty(StreamsConfig.BOOTSTRAP_SERVERS_CONFIG, "127.0.0.1:9092");
        properties.setProperty(StreamsConfig.APPLICATION_ID_CONFIG, "demo-kafka-streams");
        properties.setProperty(StreamsConfig.DEFAULT_KEY_SERDE_CLASS_CONFIG, Serdes.StringSerde.class.getName());
        properties.setProperty(StreamsConfig.DEFAULT_VALUE_SERDE_CLASS_CONFIG,
        Serdes.StringSerde.class.getName());

        // create a topology
        StreamsBuilder streamsBuilder = new StreamsBuilder();

        // input topic
        KStream<String, String> inputTopic = streamsBuilder.stream("twitter_tweets");
        KStream<String, String> filteredStream = inputTopic.filter(
            // filter for tweets which has a user of over 10000 followers
            (k, jsonTweet) -> extractUserFollowersInTweet(jsonTweet) > 10000
        );
        filteredStream.to("important_tweets");

        // build the topology
        KafkaStreams kafkaStreams = new KafkaStreams(
            streamsBuilder.build(),
            properties
        );

        // start our streams application
        kafkaStreams.start();
    }

    private static JsonParser jsonParser = new JsonParser();

    private static Integer extractUserFollowersInTweet(String tweetJson){
        // gson library
        try {
            return jsonParser.parse(tweetJson)
                .getAsJsonObject()
                .get("user")
                .getAsJsonObject()
                .get("followers_count")
                .getAsInt();
        }
        catch (NullPointerException e){
            return 0;
        }
    }
}

package com.github.simplesteph.udemy.kafka.streams;
import java.util.Properties;
import java.util.Arrays;
import org.apache.kafka.clients.consumer.ConsumerConfig;
import org.apache.kafka.common.serialization.Serdes;
import org.apache.kafka.streams.KafkaStreams;
import org.apache.kafka.streams.StreamsConfig;
import org.apache.kafka.streams.kstream.KStream;
import org.apache.kafka.streams.kstream.KStreamBuilder;
import org.apache.kafka.streams.kstream.KTable;
public class WordCountApp {
    public static void main(String[] args) {
        Properties config = new Properties();

```

```

config.put(StreamsConfig.APPLICATION_ID_CONFIG, "wordcount-application");
config.put(StreamsConfig.BOOTSTRAP_SERVERS_CONFIG, "127.0.0.1:9092");
config.put(ConsumerConfig.AUTO_OFFSET_RESET_CONFIG, "earliest");
config.put(StreamsConfig.DEFAULT_KEY_SERDE_CLASS_CONFIG, Serdes.String().getClass());
config.put(StreamsConfig.DEFAULT_VALUE_SERDE_CLASS_CONFIG, Serdes.String().getClass());
KStreamBuilder builder = new KStreamBuilder();
// 1 - stream from Kafka
KStream<String, String> textLines = builder.stream("word-count-input");
KTable<String, Long> wordCounts = textLines
    // 2 - map values to lowercase
    .mapValues(textLine -> textLine.toLowerCase())
    // can be alternatively written as:
    // .mapValues(String::toLowerCase)
    // 3 - flatmap values split by space
    .flatMapValues(textLine -> Arrays.asList(textLine.split("\\W+")))
    // 4 - select key to apply a key (we discard the old key)
    .selectKey((key, word) -> word)
    // 5 - group by key before aggregation
    .groupByKey()
    // 6 - count occurrences
    .count("Counts");
// 7 - to in order to write the results back to kafka
wordCounts.to(Serdes.String(), Serdes.Long(), "word-count-output");
KafkaStreams streams = new KafkaStreams(builder, config);
streams.start();

// shutdown hook to correctly close the streams application
Runtime.getRuntime().addShutdownHook(new Thread(streams::close));
// Update:
// print the topology every 10 seconds for Learning purposes
while(true){
    System.out.println(streams.toString());
    try {
        Thread.sleep(5000);
    } catch (InterruptedException e) {
        break;
    }
}
}

}

```

WordCount

Closing the application gracefully



- Adding a shutdown hook is key to allow for a **graceful** shutdown of the Kafka Streams application, which will help the speed of restart.
- This should be in every Kafka Streams application you create

```

// Add shutdown hook to stop the Kafka Streams threads.
// You can optionally provide a timeout to `close`
Runtime.getRuntime().addShutdownHook(new Thread(streams::close));

```

иногда при отладке полезно делать клинап

```
KafkaStreams streams = new KafkaStreams(builder, config);
// only do this in dev - not in prod
streams.cleanUp();
streams.start();

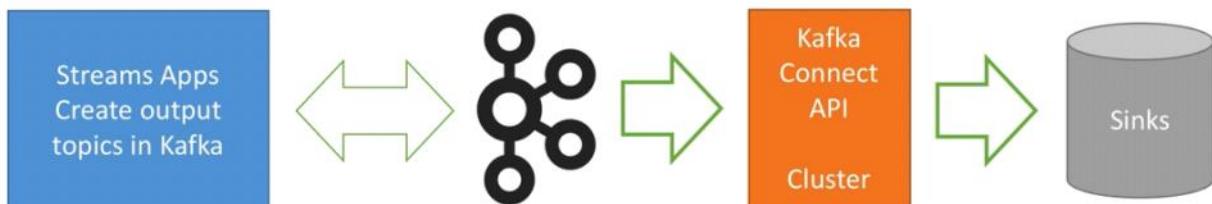
// print the topology
System.out.println(streams.toString());
```

в stream-app не рекомендуется делать что то кроме трансформаций (те писать в БД, вызывать сервисы и тп)

9 декабря 2020 г.
19:17

What if I want to write the result to an external System?

- Although it is theoretically doable to do it using [Kafka Streams library](#), it is NOT recommended
- The recommend way of doing so is Kafka Streams to transform the data and then using **Kafka Connect API** (see my other course)



java app 2

19 декабря 2020 г. 16:15

APPLICATION_ID_CONFIG одинаковый у всех кафка стрим-прог (одного DDD агрегата)

Configuring a Kafka Streams Application

- Kafka Streams configuration is specified with a `StreamsConfig` instance
 - This is typically created from a `java.util.Properties` instance
- Specify configuration parameters
 - APPLICATION_ID_CONFIG: all app instances use the same ID
 - BOOTSTRAP_SERVERS_CONFIG: where to find Kafka broker(s)
- Example:

```
Properties config = new Properties();
config.put(StreamsConfig.APPLICATION_ID_CONFIG, "simple-example");
config.put(StreamsConfig.BOOTSTRAP_SERVERS_CONFIG, "broker1:9092");
config.put(..., ...); ①
```

- ① Continue to specify configuration options

- APPLICATION_ID_CONFIG: Each stream processing application must have a unique ID. The same ID must be given to all instances of the application.



since Streams reads in from a topic and outputs to a topic, it will require Consumer and Producer configurations in addition to the Streams configurations.

SerDes SERializerDESerializer - можно дефольный незадавать, а задать отдельно для консьюмера и отдельно для продьюсера

- так как кафка-стрим может делать трансформацию одного формата в другой то десериализатор может отличаться от сериализатора, и задаваться раздельно

Serializers and Deserializers (SerDes)

- Use Serializers and Deserializers (SerDes) to convert bytes of the record to a specific type
 - SERializer
 - DESerializer
- Key SerDes can be independent from value SerDes
- There are many many built-in SerDes (e.g. `Serdes.String`, etc.)
- You can also define your own
- Configuration example:

```
config.put(StreamsConfig.DEFAULT_KEY_SERDE_CLASS_CONFIG, Serdes.String().getClass());
config.put(StreamsConfig.DEFAULT_VALUE_SERDE_CLASS_CONFIG, Serdes.Long().getClass());
```

Because Streams applications behave as both Producers and Consumers, a combination object called a SerDe is used instead of serializers and deserializers.

The SerDes defined in the Properties object are typically typed to match either the inbound or outbound messages. If the data types change due to the transforms, the SerDes can be overridden in the application code (examples will be shown later in the chapter).



Usually it is not recommended to define default, application-scoped SerDes, unless you're sure they're the only SerDes needed for the whole application!

Available SerDes

- Kafka includes a variety of SerDes in the [kafka-clients](#) Maven artifact

Data type	SerDes
byte[]	Serdess.ByteArray() , Serdess.Bytes() (Bytes wraps Java's byte[] and supports equality and ordering semantics)
ByteBuffer	Serdess.ByteBuffer()
Double	Serdess.Double()
Integer	Serdess.Integer()
Long	Serdess.Long()
String	Serdess.String()

- If you are using Avro, you can use Avro SerDes provided by the [kafka-streams-avro-serde](#) Maven artifact

пример1 раздельного задания сериализатора/десериализатора

```
final Serde<String> stringSerde = Serdes.String();
final Serde<VehiclePosition> vpSerde = getJsonSerde();

KStream<String, VehiclePosition> positions = builder
    .stream("vehicle-positions", Consumed.with(stringSerde, vpSerde));
```

. The last method to implement is the `getJsonSerde`. It's body looks like this:

```
Map<String, Object> serdeProps = new HashMap<>();
serdeProps.put("json.value.type", VehiclePosition.class);
final Serializer<VehiclePosition> vpSerializer = new KafkaJsonSerializer<>();
vpSerializer.configure(serdeProps, false);

final Deserializer<VehiclePosition> vpDeserializer = new KafkaJsonDeserializer<>();
vpDeserializer.configure(serdeProps, false);
return Serdes.serdeFrom(vpSerializer, vpDeserializer);
```

The code is somewhat self explaining. We're creating a serializer and deserializer object. They are both using the [standard Kafka Json \(De-\)Serializer](#). The class to serialize from or deserialize to is of course our `VehiclePosition` class. With this serializer/deserializer pair we then create a SerDes and return it.

пример2 раздельного задания сериализатора/десериализатора

```
StreamBuilder builder = new StreamBuilder();
```

```

StreamsBuilder builder = new StreamsBuilder();

// Construct a KStream from the input Topic "TextLinesTopic"
KStream<byte[], String> textLines = builder.stream("TextLinesTopic");

final KTable<String, Long> wordCounts = textLines
    .flatMapValues(value -> Arrays.asList(value.toLowerCase().split("\\W+")))
    // Count the occurrences of each word (record key).
    .groupBy((key, word) -> word,
        Serialized.with(Serdes.String(), Serdes.String()))
    .count("Counts");

// Write the 'KTable<String, Long>' to an output Topic
wordCounts.toStream()
    .to("WordsWithCountsTopic",
        Produced.with(Serdes.String(), Serdes.Long()));

// Run the Streams application via `start()`.

KafkaStreams streams = new KafkaStreams(builder.build(), config);
streams.start();

Runtime.getRuntime().addShutdownHook(new Thread(streams::close));
}
}

```

StreamBuilder как создать топологию

Creating the Processing Topology

- Create a `KStream` or `KTable` object from one or more Kafka Topics, using `StreamsBuilder`
- Example:

```

StreamsBuilder builder = new StreamsBuilder();

KStream<String, Long> purchases = builder.stream("PurchaseTopic");

```

The code to create a `KTable` is the same as shown for the `KStream` – just replace `KStream` with `KTable` and `builder.stream` with `builder.table`.

The `builder.stream()` starts the flow of data into the `KTable` or `KStream` by consuming messages from the specified topic. `KStreams` can specify multiple topics for `stream()`; `KTables` can only specify a single topic.

start запуск проги (билдер и конфиг подаются на вход)

Running the Application

- To start processing the stream, create a `KafkaStreams` object
 - Configure it using `StreamsBuilder`
- Then call the `start()` method
- Example:

```
KafkaStreams streams = new KafkaStreams(builder.build(), config);
streams.start();
```

(A) stateless операции трансформации

- `flatMap` will always mark the output stream for repartitioning, even if you don't change the key. So don't use those variants unless you really want to change the key, because repartitioning is a relatively expensive operation.

`filter`

Creates a new `KStream` containing only records from the previous `KStream` which meet some specified criteria

```
var smallPurchases = purchases
    .filter((key,value)->value.amount<50.0)
```

`map`

Creates a new `KStream` by transforming each element in the current stream into a different element in the new stream

```
var upper = words
    .map((key,value)->new KeyValue<>(key, value.toUpperCase()));
```

`mapValues`

Creates a new `KStream` by transforming the value of each element in the current stream into a different element in the new stream

```
var upper = words
    .mapValues(value->value.toUpperCase());
```



`filter` is the equivalent of `where` and `map` the equivalent of `select` in a SQL statement

`flatMap`

Creates a new `KStream` by transforming each element in the current stream into zero or more different elements in the new stream

<code>flatMap</code>	Creates a new <code>KStream</code> by transforming each element in the current stream into zero or more different elements in the new stream
<code>flatMapValues</code>	Creates a new <code>KStream</code> by transforming the value of each element in the current stream into zero or more different elements in the new stream

```
var pattern = Pattern.compile("\\W+", ...);

var words = textLines
    .flatMapValues(v -> Arrays.asList(pattern.split(v)));
```



A word to the difference between `map` and `mapValues` and `flatMap` and `flatMapValues`: `map` and `flatMap` will always mark the output stream for repartitioning, even if you don't change the key. So don't use those variants unless you really want to change the key, because repartitioning is a relatively expensive operation.

(Б) statefull операции трансформации

операции агрегации часто в результате дают `table`

Transforming a Stream

- Data can be transformed using a number of different operators
- Some operations result in a new `KStream` object
 - For example, `filter` or `map`
- Some operations result in a `KTable` object
 - For example, an aggregation operation

```
stream
    .groupByKey()
    .count()
```

Counts the number of instances of each key in the stream; results in a new, ever-updating `KTable`

```
stream
    .groupByKey()
    .reduce(...)
```

Combines values of the stream using a supplied Reducer into a new, ever-updating `KTable`

to как получить из стрима обратно топик

Writing Streams Back to Kafka

- Streams can be written to Kafka Topics using the `to` method
 - Example:

```
myNewStream.to("NewTopic");
```

- We often want to write to a Topic but then continue to process the data
 - Do this using the `through` method

```
myNewStream.through("NewTopic").flatMap(...)...;
```

The typical output of a Stream Topology is to a topic.

If you want to have intermediate states to work with as well, the `through()` method outputs to a topic while continuing the topology.



`through` incurs a **round-trip** (produce/consume) to Kafka, and the data read back from Kafka feeds into the next operator in the topology.

print отладка стримов

Printing A Stream's Contents

- It is sometimes useful to be able to see what a stream contains
 - Especially when testing and debugging
- Use `print()` to write the contents of the stream

```
myNewStream.print(Printed.toSysOut());
```

addShutdownHook страховка от непредвиденного завершения проги

Application Graceful Shutdown

- Allow your application to gracefully shutdown
 - For example, in response to SIGTERM
- Add a shutdown hook to stop the application

```
Runtime.getRuntime().addShutdownHook(new Thread(streams::close));
```

- After an application is stopped, Kafka migrates any tasks that had been running in this instance to available remaining instances

Join Example

- You can enrich records by merging data, i.e., joining data, from different sources
- The example below does a join based on key
 - Effectively, where `leftStream.key == rightStream.key`, use the value from `leftStream`

```
final KStream<Long, String> joinedStream =  
    leftStream.join(rightStream, (leftValue, rightValue) -> leftValue,  
    JoinWindows.of(TimeUnit.MINUTES.toMillis(5)),  
    Joined.with(Serdes.Long(), Serdes.String(), Serdes.String()));
```

how to combine streams

Converting Our Avro Example to Kafka Streams

```
import solution.model.PositionValue;  
  
// Specify Avro Serde and Schema Registry  
config.put(StreamsConfig.DEFAULT_VALUE_SERDE_CLASS_CONFIG,  
          SpecificAvroSerde.class);  
config.put(AbstractKafkaAvroSerDeConfig.SCHEMA_REGISTRY_URL_CONFIG,  
          "http://schema-registry:8081");  
  
StreamsBuilder builder = new StreamsBuilder();  
  
KStream<String, String> positions = builder.stream("vehicle-positions",  
    Consumed.with(Serdes.String(), getJsonSerde())); ①  
KStream<String, PositionValue> converted = positions  
    .mapValues(vp -> getPositionValue(vp)); ②  
  
converted.to("vehicle-positions-avro",  
    Produced.with(Serdes.String(), getPositionValueSerde()));  
  
// Continue with stream builder, start, and shutdown hook
```

- ① `getJsonSerde` creates a SerDes to convert JSON formatted data
② `getPositionValue` is the method you created in the exercise

- In a previous Hands-On Exercise, you wrote an application to write JSON formatted data into an Avro Topic
- With the Streams API in Kafka, it is very simple to convert this topic into an AVRO topic
- We assume that the `key` is of type `string` and left untouched

java app (stateless processing app)

19 декабря 2020 г. 17:13

```
public class SimpleStreamsExample {  
  
    public static void main(String[] args) throws Exception {  
        Properties config = new Properties();  
        // Give the Streams application a unique name. The name must be unique in the Kafka cluster  
        config.put(StreamsConfig.APPLICATION_ID_CONFIG, "simple-streams-example");  
        config.put(StreamsConfig.BOOTSTRAP_SERVERS_CONFIG, "broker1:9092");  
        config.put(ConsumerConfig.AUTO_OFFSET_RESET_CONFIG, "earliest");  
        // Specify default (de)serializers for record keys and for record values.  
        config.put(StreamsConfig.DEFAULT_KEY_SERDE_CLASS_CONFIG,  
                   Serdes.ByteArray().getClass());  
        config.put(StreamsConfig.DEFAULT_VALUE_SERDE_CLASS_CONFIG,  
                   Serdes.String().getClass());  
    }  
}
```

The Streams application as a whole can be launched just like any normal Java application that has a `main()` method.

Things to point out:

- Line 8 shows that the KafkaStreams type uses some of the same configurations as the standard clients
- Lines 9 and 10 show the SerDes key and value; in this example they will be the same for the inbound and outbound messages and so will not have to be overridden

```

StreamsBuilder builder = new StreamsBuilder();

// Construct a KStream from the input Topic "TextLinesTopic"
KStream<byte[], String> textLines = builder.stream("TextLinesTopic");

// Convert to upper case
KStream<byte[], String> uppercasedWithMapValues =
    textLines.mapValues(value -> value.toUpperCase());

// Write the results to a new Kafka Topic called "UppercasedTextLinesTopic".
uppercasedWithMapValues.to("UppercasedTextLinesTopic");

// Run the Streams application via `start()`
KafkaStreams streams = new KafkaStreams(builder.build(), config);
streams.start();

// Stop the application gracefully
Runtime.getRuntime().addShutdownHook(new Thread(streams::close));
}
}

```

In this example, the `textLines` KStream object is created by consuming the topic `TextLinesTopic`. `textLines` is transformed by `mapValues` to convert all of the characters in the message to uppercase, resulting in the KStream object called `uppercasedWithMapValues`. That resulting KStream is then produced to the topic `UppercasedTextLinesTopic`.



The key is `null` in this case, that's the reason why we're using the `ByteArray` SerDes for it (on the previous slide)

java app (statefull processing app)

19 декабря 2020 г. 18:06

```
public class SimpleStreamsExample {  
  
    public static void main(String[] args) throws Exception {  
        Properties config = new Properties();  
        // Give the Streams application a unique name. The name must be unique in the Kafka cluster  
        config.put(StreamsConfig.APPLICATION_ID_CONFIG, "simple-stateful-example");  
        config.put(StreamsConfig.BOOTSTRAP_SERVERS_CONFIG, "broker1:9092");  
        config.put(ConsumerConfig.AUTO_OFFSET_RESET_CONFIG, "earliest");  
        // Specify default (de)serializers for record keys and for record values.  
        config.put(StreamsConfig.DEFAULT_KEY_SERDE_CLASS_CONFIG,  
                   Serdes.ByteArray().getClass());  
        config.put(StreamsConfig.DEFAULT_VALUE_SERDE_CLASS_CONFIG,  
                   Serdes.String().getClass());  
    }  
}
```

стейтфул потому что будем использовать статефул операции агрегации

- результатом .groupBy() является KGroupedStream,
- результатом .count() является KTable,
- а также топик репартиционируется по новому ключу

```

StreamsBuilder builder = new StreamsBuilder();

// Construct a KStream from the input Topic "TextLinesTopic"
KStream<byte[], String> textLines = builder.stream("TextLinesTopic");

final KTable<String, Long> wordCounts = textLines
    .flatMapValues(value -> Arrays.asList(value.toLowerCase().split("\\W+")))
    // Count the occurrences of each word (record key).
    .groupBy((key, word) -> word,
        Serialized.with(Serdes.String(), Serdes.String()))
    .count("Counts");

// Write the `KTable<String, Long>` to an output Topic
wordCounts.toStream()
    .to("WordsWithCountsTopic",
        Produced.with(Serdes.String(), Serdes.Long()));

// Run the Streams application via `start()`.

KafkaStreams streams = new KafkaStreams(builder.build(), config);
streams.start();

Runtime.getRuntime().addShutdownHook(new Thread(streams::close));
}
}

```

In this example, the TextLinesTopic is read into the `textLines` KStream object. `textLines` is run through several stream processors which break each message into separate messages containing single words (based on whitespace) converted to lowercase (`flatMapValues()`). Each message is converted to a new message retaining the same value but also using the value as the key (`groupBy()`) before being counted based on the key and values kept in a topic called "Counts" (`count()`). `groupBy()` transparently repartitioned the data into a new internal topic before running the stateful `count()` operation to ensure that all occurrences of a given word are on the same partition. Note that because the `count()` output is stateful, a final type of KTable was specified for the generated object, `wordCounts`.

`wordCounts` is converted to a KStream (`toStream()`) before being produced to a topic called "WordWithCountsTopic". However, since the `count()` transform also changed the data types for the message, `Produced.with()` is used to override the SerDes specified in the Properties object.

... run locally

9 декабря 2020 г. 13:50

WordCount Running from IntelliJ



- Running a Kafka Streams application can be done directly from IntelliJ
- It is the preferred way to iterate in your development



Demo:

- Create the final topic using [kafka-topics](#)
- Let's run a [kafka-console-consumer](#) to the final topic
- Let's run our application from IntelliJ
- And publish some more data to our input topic using [kafka-console-producer](#)

WordCount Debugging from IntelliJ

- Let's learn how to run IntelliJ in debug mode
- This is helpful when debugging a specific line of code and understanding which statements have been happening

```
93
94 95 KafkaStreams streams = new KafkaStreams(builder, props);
    streams.start();
    System.out.println(streams.toString());
```

The screenshot shows the IntelliJ IDEA interface with the WordCountApp.java file open. A red circle highlights the break point at line 94. The code is as follows:

```
93
94 95 KafkaStreams streams = new KafkaStreams(builder, props);
    streams.start();
    System.out.println(streams.toString());
```

The debugger tool window at the bottom shows the current stack frame and variables. The stack trace includes:

```
lambda$main$2:35, WordCountApp (com.github.simplesteph.udemy.kafka.streams.wordcount)
apply(-1, 2060434592 [com.github.simplesteph.udemy.kafka.streams.WordCountApp])
apply(148, KStreamMap$1 (org.apache.kafka.streams.kstream.internals)
apply(145, KStreamMap$1$1 (org.apache.kafka.streams.processor.internals)
process$1@1, KStreamMap$KStreamMapProcessor (org.apache.kafka.streams.processor.internals)
run$2@1, ProcessorNode$1 (org.apache.kafka.streams.processor.internals)
measureLatencyNs$1@1, StreamsMetricsImpl$1 (org.apache.kafka.streams.processor.internals)
process$133, ProcessorNode$1 (org.apache.kafka.streams.processor.internals)
forward$82, ProcessorContextImpl$1 (org.apache.kafka.streams.processor.internals)
process$42, KStreamFlatMapValues$KStreamFlatMapValuesProcessor (org.apache.kafka.streams.processor.internals)
```

The variables pane shows:

- key = null
- word = "kafka"

```
~/projects/simplesteph/kafka-streams-course/word-count > master ➜ java -jar target/word-count-1.0-SNAPSHOT-jar-with-dependencies.jar
```


... run SCALING

9 декабря 2020 г. 15:01

число партиций -> число консьюмеров в группе -> число stream-приложений

на каждого консьюмера можно запустить по приложению, (тк обычно консьюмеров не может быть больше чем партиций)

WordCount Scaling our application



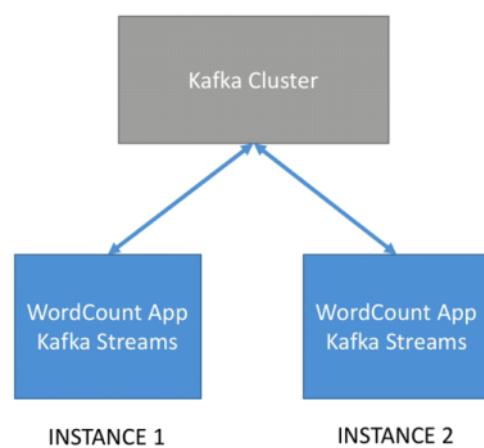
- Our input topic has **2 partitions**, therefore we can launch up to **2 instances of our application in parallel** without any changes in the code!
- This is because a Kafka Streams application relies on **KafkaConsumer**, and we saw in the Kafka Basics course that we could add consumers to a consumer group by just running the same code.
- This makes scaling super easy, without the need of any application cluster

WordCount Scaling our application



• Demo:

- Ensure the source topic has **6 partitions!** (otherwise it won't work)
- Run **two instances** of our Kafka Streams application
- Start publishing data to the source topic
- Observe our Kafka Streams application receive distinct data and still work!



- stream-прога это обычный консьюмер
- если просто запустить две stream-проги то автоматически

произойдет ребаланс

но и данные разделятся тоже по двум консьюмерам



KStream & KTable MapValues / Map

- Takes one record and produces one record
- **MapValues**
 - Is only affecting values
 - == does not change keys
 - == does not trigger a repartition
 - For KStreams and KTables
- **Map**
 - Affects both keys and values
 - Triggers a re-partitions
 - For KStreams only

```
// Java 8+ example, using lambda expressions KStream<byte[], String>
uppercased = stream.mapValues(value -> value.toUpperCase());
```



KStream & KTable Filter / FilterNot



- Takes one record and produces zero or one record
- **Filter**
 - does not change keys / values
 - == does not trigger a repartition
 - For KStreams and KTables
- **FilterNot**
 - Inverse of Filter

```
// A filter that selects (keeps) only positive numbers
KStream<String, Long> onlyPositives = stream.filter((key, value) -> value > 0);
```



KStream & KTable

FlatMapValues / FlatMap



- Takes one record and produces zero, one or more record

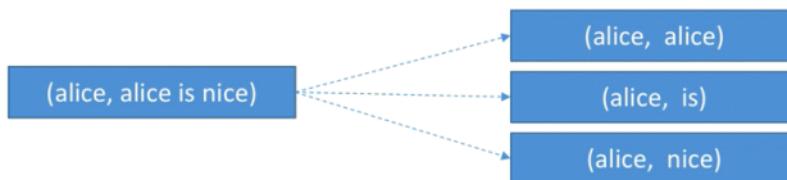
- **FlatMapValues**

- does not change keys
- == does not trigger a repartition
- For KStreams only

- **FlatMap**

- Changes keys
- == triggers a repartitions
- For KStreams only

```
// Split a sentence into words.  
words = sentences.flatMapValues(value -> Arrays.asList(value.split("\\s+")));
```

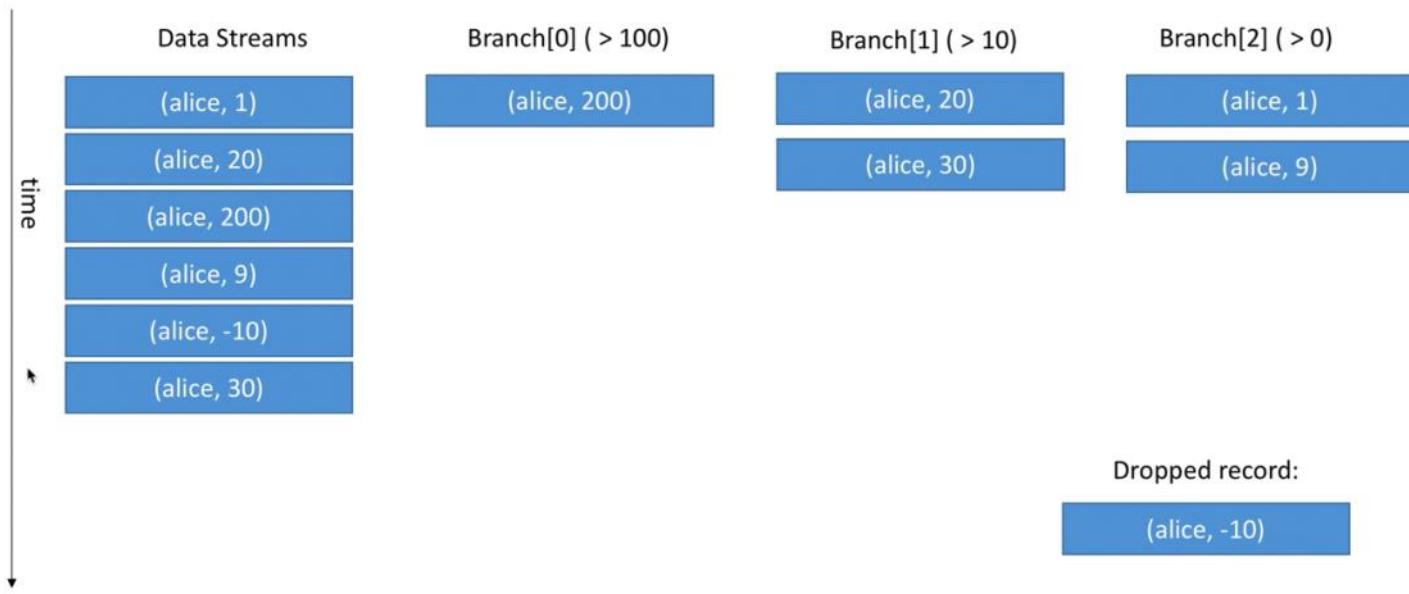


KStream Branch



- Branch (split) a KStream based on one or more predicates
- Predicates are evaluated in order, if no matches, records are dropped
- You get multiple KStreams as a result

```
KStream<String, Long>[] branches = stream.branch(  
    (key, value) -> value > 100, /* first predicate */  
    (key, value) -> value > 10, /* second predicate */  
    (key, value) -> value > 0/* third predicate */  
);
```



KStream Peek



- **Peek** allows you to apply a side-effect operation to a KStream and get the same KStream as a result.
- A side effect could be:
 - printing the stream to the console
 - Statistics collection
- Warning: It could be executed multiple times as it is side effect (in case of failures)

```
KStream<byte[], String> stream = ...;

// Java 8+ example, using lambda expressions
KStream<byte[], String> unmodifiedStream = stream.peek(
  (key, value) -> System.out.println("key=" + key + ", value=" + value));
```


... репартиционирование может случиться если
поменять ключ сообщения

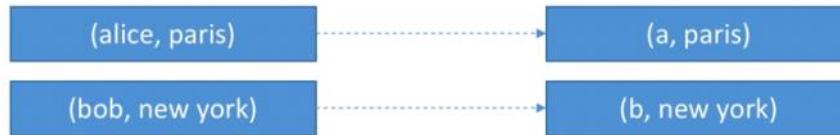
9 декабря 2020 г. 15:39



KStream SelectKey

- Assigns a new Key to the record (from old key and value)
- == marks the data for re-partitioning
- Best practice to isolate that transformation to know exactly where the partitioning happens.

```
// Use the first letter of the key as the new key
rekeyed = stream.selectKey((key, value) -> key.substring(0, 1))
```



репартиционирование медленное - поэтому просто так его
лучше не делать



Streams marked for re-partition

- As soon as an operation can possibly change the key, the stream will be marked for repartition:
 - [Map](#)
 - [FlatMap](#)
 - [SelectKey](#)
- So only use these APIs if you need to change the key, otherwise use their counterparts:
 - [MapValues](#)
 - [FlatMapValues](#)
- Repartitioning is done seamlessly behind the scenes but will incur a performance cost (read and write to Kafka)



Stateless vs Stateful Operations

- **Stateless** means that the result of a transformation only depends on the data-point you process
 - Example: a “multiply value by 2” operation is stateless because it doesn’t need memory of the past to be achieved.
 - 1 => 2
 - 300 => 600
- **Stateful** means that the result of a transformation also depends on an external information – the **state**
 - Example: a **count operation** is stateful because your app needs to know what happened since it started running in order to know the computation result
 - hello => 1
 - hello => 2

KTable GroupBy



- GroupBy allows you to perform more aggregations within a KTable
- We have used it in the previous section during our Favourite Colour Example!
- It triggers a repartition because the key changes.

```
// Group the table by a new key and key type
KGroupedTable<String, Integer> groupedTable = table.groupBy(
    (key, value) -> KeyValue.pair(value, value.length()),
    Serdes.String(), /* key (note: type was modified) */
    Serdes.Integer() /* value (note: type was modified) */ );
```

KGroupedStream / KGroupedTable Count

Count

- As a reminder, KGroupedStream are obtained after a `groupBy/groupByKey()` call on a KStream
- `Count` counts the number of record by grouped key.
- If used on KGroupedStream:
 - Null keys or values are ignored
- If used on KGroupedTable:
 - Null keys are ignored
 - Null values are treated as “delete” (= tombstones)

KGroupedStream Aggregate



- You need an initializer (of any type), an adder, a Serde and a State Store name (name of your aggregation)
- Example: Count total string length by key

```
// Aggregating a KGroupedStream (note how the value type changes from String to Long)
KTable<byte[], Long> aggregatedStream = groupedStream.aggregate(
    () -> 0L, /* initializer */
    (aggKey, newValue, aggValue) -> aggValue + newValue.length(), /* adder */
    Serdes.Long(), /* serde for aggregate value */
    "aggregated-stream-store" /* state store name */);
```

KGroupedStream / KGroupedTable Reduce



- Similar to [Aggregate](#), but the result type has to be the same as an input:
- (Int, Int) => Int (example: a * b)
- (String, String) => String (example concat(a, b))

```
// Reducing a KGroupedStream
KTable<String, Long> aggregatedStream = groupedStream.reduce(
```

```
// Reducing a KGroupedStream
KTable<String, Long> aggregatedStream = groupedStream.reduce(
    (aggValue, newValue) -> aggValue + newValue, /* adder */
    "reduced-stream-store" /* state store name */);

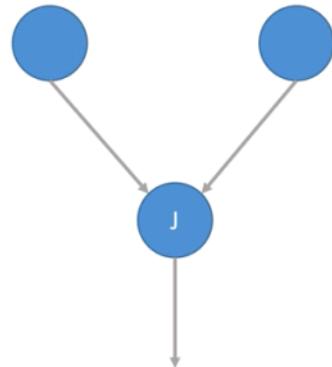
// Reducing a KGroupedTable
KTable<String, Long> aggregatedTable = groupedTable.reduce(
    (aggValue, newValue) -> aggValue + newValue, /* adder */
    (aggValue, oldValue) -> aggValue - oldValue, /* subtractor */
    "reduced-table-store" /* state store name */);
```



join - это аналог джойна по foreign key в таблицах RDBMS

What's a join?

- Joining means taking a KStream and / or KTable and creating a new KStream or KTable from it



джойн потоков можно сделать только внутри определенного временного окна

Joins KStreams and KTables



- There are 4 kind of joins (SQL-like), and the most common one will be analyzed in a further section, including behavior and usage
- See <http://docs.confluent.io/3.2.2/streams/developer-guide.html#joining> for more information

Join operands	Type	(INNER) JOIN	LEFT JOIN	OUTER JOIN
KStream-to-KStream	Windowed	Supported	Supported	Supported
KTable-to-KTable	Non-windowed	Supported	Supported	Supported
KStream-to-KTable	Non-windowed	Supported	Supported	Not Supported
KStream-to-GlobalKTable	Non-windowed	Supported	Supported	Not Supported
KTable-to-GlobalKTable	N/A	Not Supported	Not Supported	Not Supported

Table 5.2 Kafka Streams joins

Left join	Inner join	Outer join
KStream-KStream	KStream-KStream	KStream-KStream
KStream-KTable	KStream-KTable	N/A
KTable-KTable	KTable-KTable	KTable-KTable
KStream-GlobalKTable	KStream-GlobaKTable	N/A

(1) co-partitioned -число партиций в соединяемых таблицах/потоках должно совпадать
только global ktable является исключением, их можно джойнить с чем угодно

Joins Constraints

Co-partitioning of data



- These 3 joins:
 - KStream / KStream
 - KTable / KTable
 - KStream / KTable
- ...Can only happen when the data is co-partitioned. Otherwise the join won't be doable and Kafka Streams will fail with a **Runtime Error**
- That means that the **same number of partitions** is there on the stream and / or the table.
- To co-partition data, if the number of partitions is different, **write back the topics through Kafka before the join. This has a network cost**

Joins

GlobalKTable



- If your KTable data is reasonably small, and can fit on each of your Kafka Streams application, you can read it as a **GlobalKTable**
- With GlobalKTables, you can join any stream to your table even if the data doesn't have the same number of partition
- That's because the table data lives on every Streams application instance
- The downside is size on disk, but that's okay for reasonably sized dataset

(2) KEY должен быть типа VARCHAR/STRING

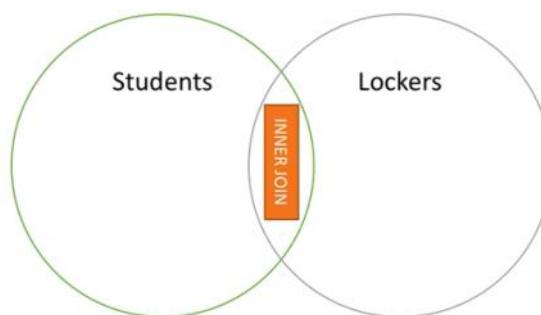
Requirements for using joins

- Joins require the columns involved in the join meet a few pre-requisites
- Same partitioning (“co-partitioned”) for join fields
- For tables
 - The KEY must VARCHAR or STRING
 - The Kafka *message key* must be the same as the *contents* of the column set in KEY
- We’ll cover work-arounds later

inner join, outer join

Inner Join

- Join the data only if it has matches in both streams of data
- Example: “Show me the Students with a Locker”

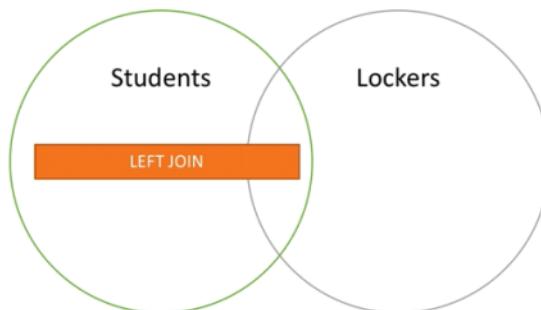


Left Join

- Join all the data from the left whether or not it has a match on the right
- Example: “Show me the Students with and without a Locker”

Left Join

- Join all the data from the left whether or not it has a match on the right
- Example: "Show me the Students with and without a Locker"



Outer Join

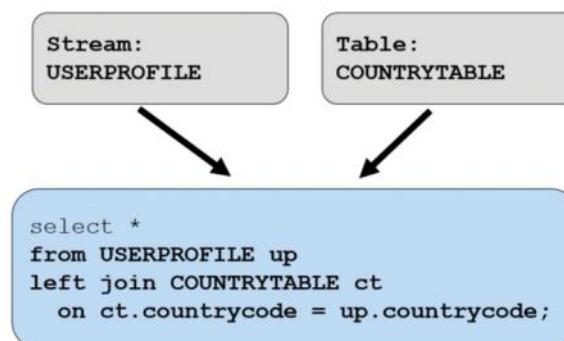


- Only available for KStream / KStream joins
- It's a left join combined with a right join.
- From the API doc, it looks like this:

stream1	stream2	result
<K1:A>		<K1:ValueJoiner(A,null)>
<K2:B>	<K2:b>	<K2:ValueJoiner(null,b)> <K2:ValueJoiner(B,b)>
	<K3:c>	<K3:ValueJoiner(null,c)>

пример

- Join
 - Stream USERPROFILE
 - Table COUNTRYTABLE
- Stream and a table ... to create a new stream.



пример

```
public static void main(String[] args) {
    Properties config = new Properties();
    config.put(StreamsConfig.APPLICATION_ID_CONFIG, "user-event-enricher-app");
    config.put(StreamsConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
    config.put(ConsumerConfig.AUTO_OFFSET_RESET_CONFIG, "earliest");
    config.put(StreamsConfig.DEFAULT_KEY_SERDE_CLASS_CONFIG, Serdes.String().getClass());
    config.put(StreamsConfig.DEFAULT_VALUE_SERDE_CLASS_CONFIG, Serdes.String().getClass());
    KStreamBuilder builder = new KStreamBuilder();
    // we get a global table out of Kafka. This table will be replicated on each Kafka Streams application
    // the key of our GlobalKTable is the user ID
    GlobalKTable<String, String> usersGlobalTable = builder.globalTable("user-table");
    // we get a stream of user purchases
    KStream<String, String> userPurchases = builder.stream("user-purchases");
    // we want to enrich that stream
    KStream<String, String> userPurchasesEnrichedJoin =
        userPurchases.join(usersGlobalTable,
            (key, value) -> key, /* map from the (key, value) of this stream to the key of the GlobalKTable */
            (userPurchase, userInfo) -> "Purchase=" + userPurchase + ",UserInfo=[" + userInfo + "]"
        );
    userPurchasesEnrichedJoin.to("user-purchases-enriched-inner-join");
    // we want to enrich that stream using a Left Join
    KStream<String, String> userPurchasesEnrichedLeftJoin =
        userPurchases.leftJoin(usersGlobalTable,
            (key, value) -> key, /* map from the (key, value) of this stream to the key of the GlobalKTable */
            (userPurchase, userInfo) -> {
                // as this is a left join, userInfo can be null
                if (userInfo != null) {
                    return "Purchase=" + userPurchase + ",UserInfo=[" + userInfo + "]";
                } else {
                    return "Purchase=" + userPurchase + ",UserInfo=null";
                }
            }
        );
    userPurchasesEnrichedLeftJoin.to("user-purchases-enriched-left-join");

    KafkaStreams streams = new KafkaStreams(builder, config);
    streams.cleanUp(); // only do this in dev - not in prod
    streams.start();
    // print the topology
    System.out.println(streams.toString());
    // shutdown hook to correctly close the streams application
    Runtime.getRuntime().addShutdownHook(new Thread(streams::close));
}
```

left join example

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

Figure 4.16 demonstrates the three possible outcomes of the outer join.

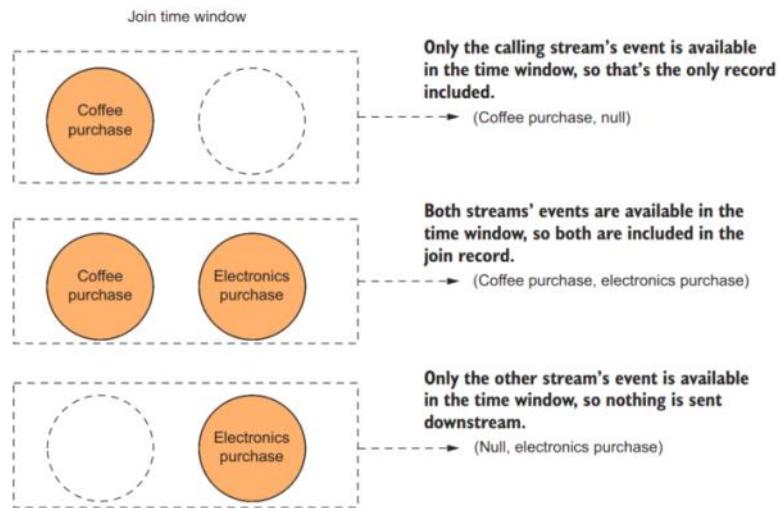


Figure 4.16 Three outcomes are possible with **outer joins**: only the calling stream's event, both events, and only the other stream's event.

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

Figure 4.17 shows the outcomes of the left-outer join.

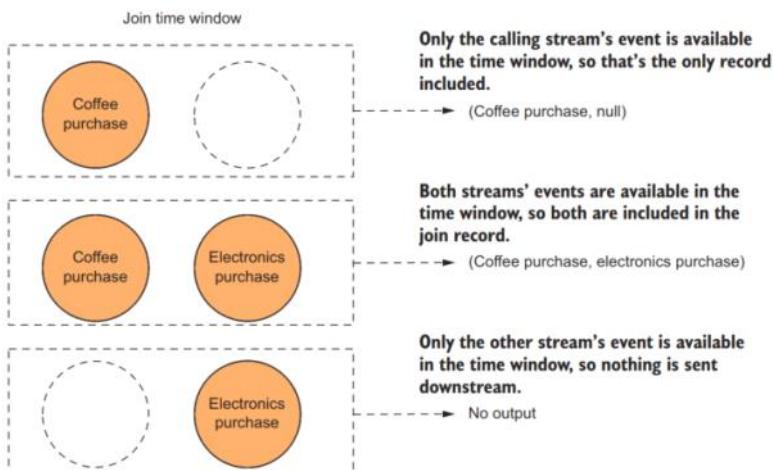


Figure 4.17 Three outcomes are possible with the **left-outer join**, but there's no output if only the other stream's record is available.

* Stream-stream join (window join)

22 февраля 2021 г. 22:03

window join - объединение окон

Оператор объединения ищет связанные события, произошедшие в течение некоего времени

- Объединения «поток — поток». Оба входных потока состоят из событий активности.
- Оператор объединения ищет связанные события, произошедшие в течение некоего времени.
- Например, это могут быть два действия одного и того же пользователя, интервал между которыми не превышает 30 минут.
- Если нужно найти связанные события в одном и том же потоке, то оба набора входных данных объединения могут принадлежать одному потоку (самообъединение).

пример

- Предположим, у нас есть функция поиска на сайте и мы хотим отслеживать последние тенденции в поиске URL. Каждый раз, когда кто-то вводит поисковый запрос, мы регистрируем событие, содержащее запрос и его результаты. Всякий раз при чём-то нажатии одного из результатов поиска мы регистрируем другое событие, записывая переход по ссылке. Чтобы рассчитать процент переходов по ссылкам для каждого URL в результатах поиска, необходимо объединить события поиска и перехода по ссылке, связанные одинаковым ID сессии. Подобный анализ часто используется в рекламных системах [85].
- Можно выбрать для объединения подходящее окно, например, объединить нажатие с поиском, если между ними прошло не более часа.
- Чтобы реализовать этот тип объединения, поточный процессор должен поддерживать состояния, например все события, произошедшие за последний час и индексированные по ID сессии. Всякий раз, когда происходит событие поиска или нажатия, оно добавляется к соответствующему индексу. Потоковый процессор каждый раз проверяет оба индекса с целью узнать, не получено ли уже другое событие для того же ID сессии. При поступлении соответствующего события процессор генерирует событие, в котором указывается, какой результат поиска был выбран. Если событие поиска истекло, а соответствующее событие нажатия не было получено, то генерируется событие, в котором сообщается, что пользователь не выбрал ни один из результатов поиска.

* Stream-table join (stream enrichment)

22 февраля 2021 г. 22:07

stream enrichment

- пример пакетной задачи, объединяющей два набора данных: набор событий активности пользователя и БД пользовательских профилей (см. рис. 10.2). Естественно представлять события активности как поток и постоянно выполнять в потоковом процессоре одно и то же объединение: на входе поток событий активности, содержащих ID пользователя, а на выходе — поток событий активности, в которых идентификатор дополнен информацией о его профиле. Этот процесс иногда называют обогащением событий активности информацией из базы данных
- Объединение «поток — таблица». Один входной поток состоит из событий активности, а второй представляет собой журнал изменений базы данных. В журнале хранится локальная копия БД. Для каждого события активности оператор объединения делает запрос в базу и выводит событие активности, обогащенное информацией из БД.

таблица может храниться локально в памяти

- Чтобы выполнить это объединение, потоковый процесс должен просматривать события активности по одному, находить ID пользователя в базе данных и добавлять в событие активности информацию о профиле этого человека
- Другой подход заключается в том, чтобы загрузить в потоковый процессор копию базы и запрашивать ее локально, без передачи данных по сети. Это очень похоже на хеш-объединения, описанные в подразделе «Объединения на этапе сопоставления» раздела 10.2: локальная копия БД может храниться в памяти в виде хеш-таблицы, если она достаточно мала, или же в виде индекса на локальном диске.

сама таблица должна постоянно обновляться

- В отличие от пакетных задач, где используется моментальный снимок базы данных на момент ввода исходных данных, потоковый процессор работает долгое время, и содержимое базы, очевидно, со временем изменится. В то время локальная копия БД потокового процессора должна обновиться.
- Эту проблему можно решить путем сбора данных об изменениях: потоковый процессор может подписаться на журнал изменений пользовательских профилей в базе данных, а также на поток событий активности. При создании или изменении профиля потоковый процессор обновит свою локальную копию базы данных. Таким образом, мы получаем объединение двух потоков: событий активности и обновлений профиля

На практике объединение «поток — таблица» очень похоже на объединение «поток — поток»

- главное различие заключается в том, что для потока изменений таблицы объединение использует окно, начало которого совпадает с «началом времени» (концептуально бесконечное окно), причем новые версии записей перезаписывают старые.
- Для входных данных потока объединение может вообще не поддерживать окно.

* Table-table join (materialized view maintenance)

22 февраля 2021 г. 22:12

materialized view maintenance

Оба входных потока являются изменениями для таблиц базы данных

- Объединение «таблица — таблица». Оба входных потока являются изменениями базы данных.
- В этом случае каждое изменение из одной таблицы объединяется с последним состоянием из другой.
- Результатом является поток изменений, который передается в материализованное представление объединения двух таблиц.

пример

- когда пользователь хочет просмотреть свою ленту сообщений, слишком накладно перебирать всех людей, которых он отслеживает, находить их последние твиты и объединять их
- Лучше применять кэш ленты сообщений: вариант папки Входящие для каждого пользователя, в которую записываются твиты по мере их отправки, вследствие чего чтение ленты сообщений выполняется за один просмотр. Для реализации и поддержки такого кэша нужна следующая обработка событий

представление запроса, объединяющего две таблицы (твиты и подписки)

- нужны потоки событий для твитов (отправка и удаление) и отношения подписки (отслеживание и устранение).
- Процесс потока должен поддерживать базу данных, содержащую набор подписчиков для каждого пользователя, чтобы знать, какие ленты сообщений необходимо обновлять при поступлении нового твита
- Объединение потоков прямо соответствует объединению таблиц в данном запросе. Ленты сообщений — фактически кэш результата этого запроса, который обновляется всякий раз при изменении соответствующих таблиц

```
SELECT follows.follower_id AS timeline_id,
       array_agg(tweets.* ORDER BY tweets.timestamp DESC)
  FROM tweets
 JOIN follows ON follows.followee_id = tweets.sender_id
 GROUP BY follows.follower_id
```

* НИЗКОУРОВНЕВЫЕ ОПЕРАЦИИ

1 февраля 2021 г. 13:31

пример трансформации

onenote:///C:/Users/trans/Qsync/vova_from_onenote/tf_algonote_v1\SYSTEMDESIGN\KAFKA \ LOCAL%20STATE%20STORE.one#сохраняем%20состояние%20в%20трансформации§ion-id={720F6F5D-93F8-4266-949DE246C971A22}&page-id={B528C761-5DD0-4DBC-ACDB-441CE38593A2}&end

Создание узла-источника

Listing 6.1 Creating the beer application source node

```
topology.addSource(LATEST,  
                   purchaseSourceNodeName,  
                   new UsePreviousTimeOnInvalidTimestamp(),  
                   stringDeserializer,  
                   beerPurchaseDeserializer,  
                   Topics.POPS_HOPS_PURCHASES.topicName())
```

Specifies the offset reset to use
Specifies the name of this node
Sets the key deserializer
Sets the value deserializer
Specifies the name of the topic to consume data from
Specifies the TimestampExtractor to use for this source

Добавление узла-обработчика

- Метод Topology.addProcessor принимает в качестве второго параметра экземпляр интерфейса ProcessorSupplier, но, поскольку ProcessorSupplier — интерфейс с одним методом, его можно заменить лямбда-выражением.

Listing 6.2 Adding a processor node

```
BeerPurchaseProcessor beerProcessor =  
    new BeerPurchaseProcessor(domesticSalesSink, internationalSalesSink);  
  
topology.addSource(LATEST,  
                   purchaseSourceNodeName,  
                   new UsePreviousTimeOnInvalidTimestamp(),  
                   stringDeserializer,  
                   beerPurchaseDeserializer,  
                   Topics.POPS_HOPS_PURCHASES.topicName())  
    .addProcessor(purchaseProcessor,  
                 () -> beerProcessor,  
                 purchaseSourceNodeName);
```

Adds the processor defined above
Names the processor node
Specifies the name of the parent node or nodes

```

builder.addSource(LATEST,
    purchaseSourceNodeName, ←
    new UsePreviousTimeOnInvalidTimestamp()
    stringDeserializer,
    beerPurchaseDeserializer,
    "pops-hops-purchases");
}

builder.addProcessor(purchaseProcessor,
    () -> beerProcessor,
    purchaseSourceNodeName);

```

The **name** of the source node (above) is used for the **parent name** of the processing node (below). This establishes the parent-child relationship, which directs data flow in Kafka Streams.

Figure 6.2 Wiring up parent and child nodes in the Processor API

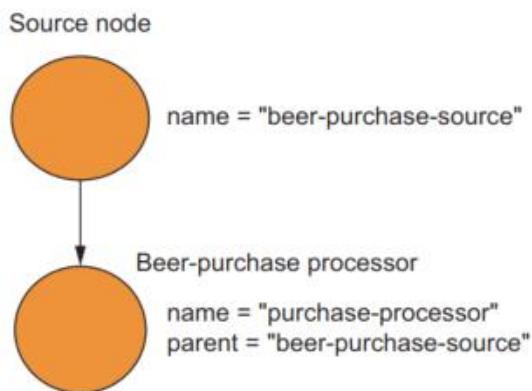


Figure 6.3 The Processor API topology so far, including node names and parent names

Именно в методе `Processor.process()` мы производим действия над проходящими через топологию записями.

- В интерфейсе `Processor` имеются методы `init()`, `process()`, `punctuate()` и `close()`.
- `Processor` — основная движущая сила логики любого работающего с записями потокового приложения.
- В примерах мы в основном будем использовать класс `AbstractProcessor`, так что переопределять станем только необходимые нам методы. Класс `AbstractProcessor` сам инициализирует объект `ProcessorContext`, так что переопределять метод `init()` не нужно, разве что вам понадобится выполнить какие-либо дополнительные настройки.

Listing 6.3 BeerPurchaseProcessor

```
public class BeerPurchaseProcessor extends
    AbstractProcessor<String, BeerPurchase> {

    private String domesticSalesNode;
    private String internationalSalesNode;

    public BeerPurchaseProcessor(String domesticSalesNode,
                                String internationalSalesNode) {
        this.domesticSalesNode = domesticSalesNode;
        this.internationalSalesNode = internationalSalesNode; ←
    }

    @Override
    public void process(String key, BeerPurchase beerPurchase) { ←

        Currency transactionCurrency = beerPurchase.getCurrency();

        if (transactionCurrency != DOLLARS) { ←
            BeerPurchase dollarBeerPurchase;
            BeerPurchase.Builder builder =
                BeerPurchase.newBuilder(beerPurchase);
            double internationalSaleAmount = beerPurchase.getTotalSale();
            String pattern = "###.##";
            DecimalFormat decimalFormat = new DecimalFormat(pattern);
            builder.currency(DOLLARS);
            builder.totalSale(Double.parseDouble(decimalFormat.
                format(transactionCurrency
                    .convertToDollars(internationalSaleAmount)))); ←
            dollarBeerPurchase = builder.build();
            context().forward(key,
                dollarBeerPurchase, internationalSalesNode); ←
        } else {
            context().forward(key, beerPurchase, domesticSalesNode);
        }
    }
}
```

Sends records for domestic sales to the domestic child node

Sets the names for different nodes to forward records to

The process() method, where the action takes place

Converts international sales to US dollars

Uses the ProcessorContext (returned from the context() method) and forwards records to the international child node

Listing 6.7 process() implementation

```
@Override
public void process(String symbol, StockTransaction transaction) {
    StockPerformance stockPerformance = keyValueStore.get(symbol); ←
    if (stockPerformance == null) {
        stockPerformance = new StockPerformance(); ←
    }
    stockPerformance.updatePriceStats(transaction.getSharePrice());
    stockPerformance.updateVolumeStats(transaction.getShares()); ←
    stockPerformance.setLastUpdateSent(Instant.now());
    keyValueStore.put(symbol, stockPerformance); ←
}
```

Updates the price statistics for this stock

Sets the timestamp of the last update

Retrieves previous performance stats, possibly null

Creates a new StockPerformance object if one isn't in the state store

Updates the volume statistics for this stock

Places the updated StockPerformance object into the state store

Добавление узла-стока

Listing 6.4 Adding a sink node

```
topology.addSource(LATEST,
                    purchaseSourceNodeName,
                    new UsePreviousTimeOnInvalidTimestamp(),
                    stringDeserializer,
                    beerPurchaseDeserializer,
                    Topics.POPS_HOPS_PURCHASES.topicName())
    .addProcessor(purchaseProcessor,
                  () -> beerProcessor,
                  purchaseSourceNodeName)
    .addSink(internationalSalesSink,
             "international-sales",
             stringSerializer,
             beerPurchaseSerializer,
             purchaseProcessor)
    .addSink(domesticSalesSink,
             "domestic-sales",
             stringSerializer,
             beerPurchaseSerializer,
             purchaseProcessor);
```

The diagram illustrates the addition of a sink node in a Kafka Streams topology. It shows the flow from source to processor to two sinks. Annotations explain various components:

- Serializer for the key**: Points to the first parameter of `.addSink`.
- Parent node for this sink**: Points to the parent node before the first sink.
- Name of the sink**: Points to the second parameter of `.addSink`.
- The topic this sink represents**: Points to the third parameter of `.addSink`.
- Serializer for the value**: Points to the fourth parameter of `.addSink`.
- Parent node for this sink**: Points to the parent node before the second sink.
- Name of the sink**: Points to the second parameter of `.addSink`.
- The topic this sink represents**: Points to the third parameter of `.addSink`.

три метода, с помощью которых можно интегрировать созданную с помощью API узлов-обработчиков функциональность

- Интеграция API узлов-обработчиков и API Kafka Streams
- API Kafka Streams предоставляет три метода, с помощью которых можно интегрировать созданную с помощью API узлов-обработчиков функциональность: `KStream.process`, `KStream.transform` и `KStream.transformValues`. Вам должен быть знаком этот подход, поскольку вы уже работали с интерфейсом `ValueTransformer`
- Метод `KStream.process` создает концевой узел,
- в то время как методы `KStream.transform` и `KStream.transformValues` возвращают новый экземпляр `KStream`, благодаря чему вы можете продолжать добавлять новые узлы-обработчики к данному узлу. Отмету также, что методы преобразования сохраняют состояние, поэтому при их использовании необходимо передавать название хранилища состояния. Поскольку `KStream.process` создает концевой узел, то обычно используется или `KStream.transform`, или `KStream.transformValues`
- там вы можете заменить свой экземпляр `Processor` на `Transformer`. Основное различие между этими двумя интерфейсами состоит в том, что метод, в котором происходят основные действия у `Processor`, — `process()` с возвращаемым типом `void`, а у `Transformer` это `transform()` с возвращаемым типом `R`. Оба интерфейса предоставляют одинаковую семантику пунктуации.
- В большинстве случаев для замены `Processor` достаточно переместить код логики из метода `Processor.process` в метод `Transformer.transform`. Нужно учесть только необходимость возврата значения, но можно вернуть пустое значение и отправлять результаты далее с помощью метода `ProcessorContext.forward`.
- The transformer returns a value: in this case, it returns a null, which is filtered out, and you use the `ProcessorContext.forward` method to send multiple values downstream. If you wanted to return multiple values instead, you'd return a `List<KeyValue<K,V>>` and then attach a `flatMap` or `flatMapValues` to send individual

records downstream. An example of this can be found in `src/main/java/bbejeck/chapter_6/StockPerformanceStreamsAndProcessorMultipleValuesApplication.java`. To complete the replacement of a Processor instance, you'd plug in the Transformer (or ValueTransformer) instance using the `KStream.transform` or `KStream.transformValues` method.

kafkaStreams.setUncaughtExceptionHandler

2 февраля 2021 г. 14:03

Обработчик неперехваченных исключений

Для обработки подобных неожиданных ошибок Kafka Streams предоставляет метод KafkaStreams.setUncaughtExceptionHandler

Listing 7.10 Basic uncaught exception handler

```
kafkaStreams.setUncaughtExceptionHandler((thread, exception) -> {
    CONSOLE_LOG.info("Thread [" + thread + "]"
        ↗ encountered [" + exception.getMessage() + "]");
});
```

настройки

2 февраля 2021 г. 21:10

настройки для устойчивости к отказам серверов

Благодаря заданию таких настроек в случае аварийного останова всех брокеров кластера Kafka ваше приложение Kafka Streams останется работоспособным и будет готово возобновить работу после восстановления брокеров.

- Set Producer.NUM_RETRIES to Integer.MAX_VALUE.
- Set Producer.REQUEST_TIMEOUT to 305000 (5 minutes).
- Set Producer.BLOCK_MS_CONFIG to Integer.MAX_VALUE.
- Set Consumer.MAX_POLL_CONFIG to Integer.MAX_VALUE.

Listing A.1 Setting properties for resilience to broker outages

```
Properties props = new Properties();
props.put(StreamsConfig.producerPrefix(
    ProducerConfig.RETRIES_CONFIG), Integer.MAX_VALUE);
props.put(StreamsConfig.producerPrefix(
    ProducerConfig.MAX_BLOCK_MS_CONFIG), Integer.MAX_VALUE);
props.put(StreamsConfig.REQUEST_TIMEOUT_MS_CONFIG, 305000);
props.put(StreamsConfig.consumerPrefix(
    ConsumerConfig.MAX_POLL_INTERVAL_MS_CONFIG), Integer.MAX_VALUE);
```

чтобы при ошибках десериализации клиент продолжил работу

- Kafka работает с байтовыми массивами ключей и значений, и для их использования ключи и значения нужно десериализовать. Именно поэтому для всех узлов-источников и узлов-стоков требуются объекты Serde. Появление испорченных данных во время обработки записей вполне возможно. Поэтому для указания, как поступать с этими ошибками десериализации, в Kafka Streams существуют параметры default.serialization.exception.handler и StreamsConfig.DEFAULT_DESERIALIZATION_EXCEPTION_HANDLER_CLASS_CONFIG
- Значение по умолчанию для этих параметров — класс org.apache.kafka.streams.errors.LogAndFailExceptionHandler, выполняющий, как можно догадаться из названия, журналирование ошибки. При такой настройке генерация исключения десериализации приведет к сбою (аварийному останову) приложения Kafka Streams. Другой класс, org.apache.kafka.streams.errors.LogAndContinueExceptionHandler, заносит ошибку в журнал, но приложение Kafka Streams при этом продолжает работать.
Вы можете реализовать свои собственные обработчики исключений десериализации путем создания класса, реализующего интерфейс DeserializationExceptionHandler

Listing A.2 Setting a deserialization handler

```
Properties props = new Properties();
props.put(StreamsConfig.DEFAULT_DESERIALIZATION_EXCEPTION_HANDLER_CLASS_CONFIG,
    LogAndContinueExceptionHandler.class);
```

СКОЛЬКО НУЖНО ПОТОКОВ ВЫПОЛНЕНИЯ И СКОЛЬКО — ЭКЗЕМПЛЯРОВ Kafka Streams.

- Как вы помните из главы 3, Kafka Streams создает объекты StreamTask по числу секций входного топика (-ов). В нашем первом примере для простоты мы рассмотрим входной топик из 12 секций.
- В случае 12 входных секций Kafka Streams создаст 12 задач. Допустим, нам нужно по одной задаче на поток выполнения. Можно использовать один экземпляр с 12 потоками выполнения, но у такого подхода есть недостаток: в случае сбоя машины, на которой запущено приложение Kafka Streams, вся потоковая обработка прекратится.
- Но если запускать экземпляры с четырьмя потоками выполнения каждый, то каждый экземпляр будет обрабатывать четыре входные секции. Преимущество такого подхода: в случае сбоя одного из экземпляров Kafka Streams будет запущена перебалансировка и четыре задачи неработающего экземпляра будут перераспределены по двум остальным, таким образом, оставшиеся экземпляры будут обрабатывать по шесть задач каждый. Кроме того, после возобновления работы остановленного экземпляра произойдет еще одна перебалансировка и все три экземпляра снова будут обрабатывать по четыре задачи каждый.
- Важно учесть, что Kafka Streams определяет количество создаваемых задач на основе максимального количества секций из всех входных топиков. В случае одного топика с 12 секциями число задач будет равно 12; но если входных топиков у вас четыре, по три секции каждый, то задач окажется три, каждая из которых будет отвечать за обработку четырех секций.
-

если потоков выполнения больше, чем число задач, то оставшиеся потоки будут простоять

- если запустить в вышеописанном примере с тремя экземплярами Kafka Streams четвертый экземпляр с четырьмя потоками выполнения, то после перебалансировки у вас окажется четыре простояющих потока выполнения (16 потоков выполнения, но лишь 12 задач).

явная Конфигурация RocksDB

- Для задания пользовательских настроек RocksDB создайте класс, реализующий интерфейс RocksDBConfigSetter, и укажите название этого класса в конфигурации своего приложения Kafka Streams, в параметре StreamsConfig.ROCKSDB_CONFIG_SETTER_CLASS_CONFIG. Узнать больше о возможностях настройки RocksDB вы можете из руководства по настройке RocksDB (RocksDB Tuning Guide, <http://mng.bz/l88k>).

явная заблаговременная перебалансировка

- В Kafka Streams при всякой операции, способной потенциально изменить ключ тар — например, transform или groupBy, — в классе StreamsBuilder устанавливается внутренний флаг, указывающий на необходимость повторного секционирования. При этом выполнение операций тар или transform не приведет автоматически к созданию временного топика повторного секционирования; но операция повторного секционирования будет запущена,

- как только вы добавите любую операцию, использующую обновленный ключ.
- Хотя этот шаг является обязательным (см. главу 4), в некоторых случаях лучше самостоятельно выполнить повторное секционирование заблаговременно
 - Решение этой проблемы очень простое: **после вызова метода тар необходимо сразу же вызвать операцию through для секционирования данных.** При этом последующие вызовы groupByKey не запустят повторное секционирование данных, поскольку оператор groupByKey не устанавливает флаг потребности в повторном секционировании.

явное управление внутренними топиками

- Есть два варианта управления внутренними топиками.
- Но помните, что размер топика по умолчанию не ограничен, а время сохранения по умолчанию равно одной неделе, так что имеет смысл установить подходящие значения параметров retention.bytes и retention.ms.
- Помимо этого, для журналов изменений, служащих в качестве резервной копии хранилищ состояния с большим количеством уникальных ключей, можно установить параметр cleanup.policy равным compact, delete, чтобы размер топика не слишком рос.

Во-первых, можно задавать настройки непосредственно при создании хранилищ состояния с помощью метода `StoreBuilder.withLoggingEnabled` или `Materialized.withLoggingEnabled`.

- Какой метод использовать — зависит от способа создания хранилища состояния. Оба метода принимают в качестве параметра объект `Map<String, String>` со свойствами топика. Пример их использования можно найти в файле `src/main/java/bbejeck/chapter_7/CoGroupingListeningExampleApplication`

Второй вариант управления внутренними топиками — указание настроек для них в процессе настройки приложения Kafka Streams:

- В случае подхода со `StreamsConfig.topicPrefix` указанные настройки используются глобально, для всех внутренних топиков
- При этом настройки, задаваемые при создании хранилища состояния, имеют приоритет над настройками, указываемыми с помощью класса `StreamsConfig`.

```
Properties props = new Properties();
// other properties set here
props.put(StreamsConfig.topicPrefix("retention.bytes"), 1024 * 1024);
props.put(StreamsConfig.topicPrefix("retention.ms"), 3600000);
```

перезапуска приложения Kafka Streams и повторной обработки данных или в процессе разработки, или после установки обновления кода

- Для этой цели Kafka Streams предоставляет **сценарий kafka-streams-application-reset.sh**, расположенный в подкаталоге `bin` каталога, в который установлен Kafka.
- У данного сценария есть один обязательный параметр: идентификатор приложения Kafka Streams. Он предлагает несколько возможностей (если вкратце): сбрасывать входящие топики до самого раннего из имеющихся смещений, сбрасывать промежуточные топики до последнего смещения и удалять любые внутренние топики. Отмету, что при первом запуске приложения необходимо вызвать метод `KafkaStreams.cleanUp`, чтобы удалить локальное состояние, сохраненное с предыдущих запусков.

Для полной очистки локального состояния можно воспользоваться методом KafkaStreams.cleanUp

- либо перед вызовом метода KafkaStreams.start, либо после вызова KafkaStreams.stop. Вызов метода KafkaStreams.cleanUp в любой другой момент приведет к ошибке.

badTopic

3 февраля 2021 г. 18:53

badTopic - это тема Kafka, в которую мы будем писать ошибки

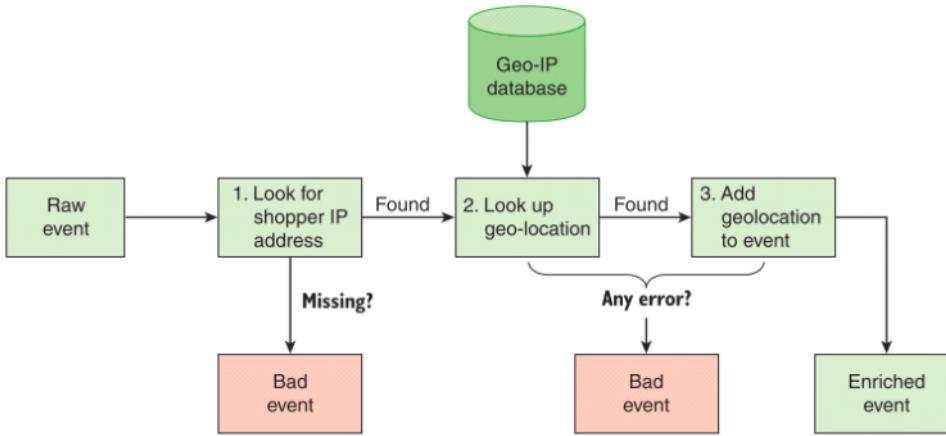


Figure 3.9 Our single-event processor attempts to enrich the raw event with the geolocation as looked up in the MaxMind database; if anything goes wrong, an error is written out instead.

3.4.2 Updating our main function

Head back to `src/main/java/nile/StreamApp.java` and make the additions set out in the following listing.

Listing 3.7 StreamApp.java

```
package nile;
import java.io.*;
import com.maxmind.geoip2.DatabaseReader;

public class StreamApp {

    public static void main(String[] args) throws IOException {
        String servers      = args[0];
        String groupId      = args[1];
        String inTopic       = args[2];
        String goodTopic    = args[3];
        String badTopic     = args[4];
        String maxmindFile  = args[5];

        Initialize the MaxMind geo-IP database.           | Initialize the MaxMind geo-IP database can throw an exception.
        Consumer consumer = new Consumer(servers, groupId, inTopic);
        DatabaseReader maxmind = new DatabaseReader.Builder(new File(maxmindFile)).build();
        FullProducer producer = new FullProducer(
            servers, goodTopic, badTopic, maxmind);
        consumer.run(producer);
    }
}
```

Initialize the MaxMind geo-IP database. | Initialize the MaxMind geo-IP database can throw an exception.

New command-line arguments

Initialize the FullProducer with both outbound Kafka topics and the MaxMind geo-IP database.

Wait a few seconds and you should start to see the validation failures stream into the `bad-events` topic:

```
{"error": "JsonParseException: Unrecognized token 'not':\nwas expecting 'null', 'true', 'false' or NaN\nat [Source: not json; line: 1, column: 4]"}\n{"error": "NullPointerException: null"}\n{"error": "shopper.ipAddress missing"}
```