7 Streams and state

This chapter covers

- **Adding stateful operations to Kafka Streams**
- **Using state stores in Kafka Streams**
- **Enriching event streams with joins**
- **Learning how timestamps drive Kafka Streams**

In the last chapter, we dove headfirst into the Kafka Streams DSL and built a processing topology to handle streaming requirements from purchase activity. Although you built a nontrivial processing topology, it was one dimensional in that all transformations and operations were stateless. You considered each transaction in isolation, without any regard to other events occurring at the same time or within certain time boundaries, either before or after the transaction. Also, you only dealt with individual streams, ignoring any possibility of gaining additional insight by joining streams together.

In this chapter, you'll extract the maximum amount of information from the Kafka Streams application. To get this level of information, you'll need to use state. State is nothing more than the ability to recall information you've seen before and connect it to current information. You can utilize state in different ways. We'll look at one example when we explore the stateful operations, such as the accumulation of values, provided by the Kafka Streams DSL.

We'll get to another example of using state when we'll discuss the joining of streams. Joining streams is closely related to the joins performed in database operations, such as joining records from the employee and department tables to generate a report on who staffs which departments in a company.

We'll also define what the state needs to look like and what the requirements are for using state

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/kafka-streams-in-action-second-edition/discussion

when we discuss state stores in Kafka Streams. Finally, we'll weigh the importance of timestamps and look at how they can help you work with stateful operations, such as ensuring you only work with events occurring within a given time frame or helping you work with data arriving out of order.

7.1 Stateful vs stateless

Before we go on with examples, let's provide a description of the difference between stateless and stateful. In a stateless operation there is no additional information retrieved, what's present is enough to complete the desired action. On the other hand, a stateful operation is more complex because it involves keeping the state of previous event. A basic example of a stateful operation is an aggregation.

For example, consider this function:

Listing 7.1 Stateless function example

```
public boolean numberIsOnePredicate (Widget widget) {

    return widget.number == 1;
}
```

Here all the `Widget` object contains all the information needed to execute the predicate, there's no need to lookup or store data. Now let's take a look at an example of a stateful function

Listing 7.2 Stateful function example

```
public int count(Widget widget) {

  int widgetCount = hashMap.compute(widget.id,
   (key, value) -> (value == null) ? 1 : value + 1)

  return widgetCount;
}
```

Here in the `count` function, we are computing the total of widgets with the same id. To perform the count we first must look up the current number by id, increment it, and then store the new number. If no number is found, we go ahead and provide an initial value, a 1 in this case.

While this is a trivial example of using state, the principals involved are what matter here. We are using a common identifier across different objects, called a key, to store and retrieve some value type to track a given state that we want to observe. Additionally, we use an initializing function to produce a value when one hasn't been calculated yet for a given key.

These are the core steps we're going to explore and use in this chapter, although it will be far more robust than using the humble `HashMap`!

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/kafka-streams-in-action-second-edition/discussion

7.2 Adding stateful operations to Kafka Streams

So the next question is why you need to use state when processing an event stream? The answer is any time you need to track information or progress across related events. For example consider a Kafka Streams application tracking the progress of players in an online poker game. Participants play in rounds and their score from each round is transmitted to a server then reset to zero for the start of the next round. The game server the produces the players score to a topic.

A stateless event stream will give you the opportunity to work with the current score from the latest round. But for tracking their total, you'll need to keep the state of all their previous scores.

This scenario leads us to our first example of a stateful operation in Kafka Streams. For this example we're going to use a reduce. A reduce operation takes multiple values and reduces or merges them into a single result. Let's look at an illustration to help understand how this process works:

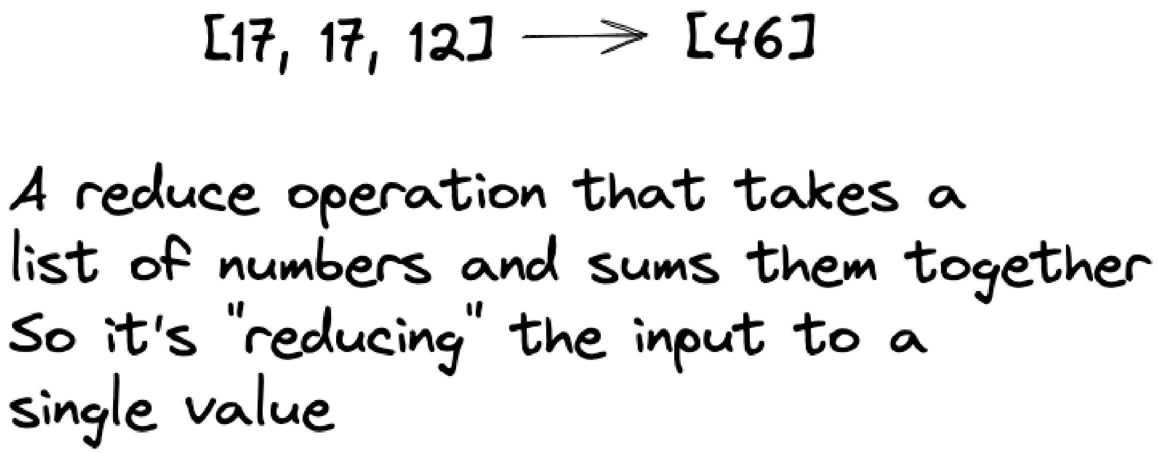
[17, 17, 12] → [46]
A reduce operation that takes a list of numbers and sums them together So it's "reducing" the input to a single value

Figure 7.1 A reduce takes several inputs and merges them into a single result of the same type

As you can see in the illustration, the reduce operation takes five numbers and "reduces" them down to a single result by summing the numbers together. So Kafka Streams takes an unbounded stream of scores and continues to sum them per player. At this point we've described the reduce operation itself, but there's some additional information we need to cover regarding how Kafka Streams sets up to perform the reduce.

When describing our online poker game scenario, I mentioned that there are individual players, so it stands to reason that we want to calculate total scores for each ***individual***. But we aren't guaranteed the order of the incoming player scores, so we need the ability to group them. Remember Kafka works with key-value pairs, so we'll assume the incoming records take the form of playerId-score for the key-value pair.

So if the key is the player-id, then all Kafka Streams needs to do is bucket or group the scores by the id and you'll end up with the summed scores per player. It will probably be helpful for us to view an illustration of the concept:

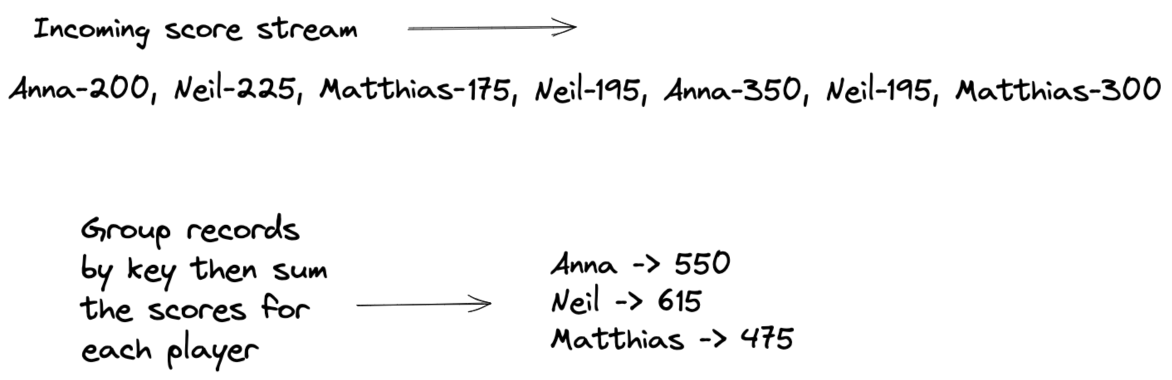
Incoming score stream
Anna-200, Neil-225, Matthias-175, Neil-195, Anna-350, Neil-195, Matthias-300
Group records by key then sum the scores for each player
Anna -> 550
Neil -> 615
Matthias -> 475

Figure 7.2 Grouping the scores by player-id ensures we only sum the scores for the individual players

So by grouping the scores by player-id, you are guaranteed to only sum the scores for each player. This group-by functionality in Kafka Streams is similar to the SQL group-by when performing aggregation operations on a database table.

NOTE **At this point going forward, I'm not going to show the basic setup code needed i.e. creating the `StreamBuilder` instance and serdes for the record types. You've learned in the previous chapter how these components fit into an application, so you can refer back if you need to refresh your memory.**

Now let's see the reduce in action with Kafka Streams

Listing 7.3 Performing a reduce in Kafka Streams to show running total of scores in an online poker game

```
KStream<String, Double> pokerScoreStream = builder.stream("poker-game",
        Consumed.with(Serdes.String(), Serdes.Double()));

pokerScoreStream
        .groupByKey()  ❶
        .reduce(Double::sum,   ❷
                Materialized.with(Serdes.String(), Serdes.Double()))
        .toStream()  ❸
        .to("total-scores",
                Produced.with(Serdes.String(), Serdes.Double()));  ❹
```

❶ Grouping by key so that scores are calculated by individual keys

❷ Reducer as a method reference

❸ Converting the KTable to a stream

❹ Writing the results out to a topic

This Kafka Streams application results in key-value pairs like "Neil, 650" and it's a continual stream of summed scores, continually updated.

Looking over the code you can see you first perform a `groupByKey` call. It's important to note that grouping by key is a prerequisite for stateful aggregations in Kafka Streams. So what do you do when there is no key or you need to derive a different one? For the case of using a different key, the `KStream` interface provides a `groupBy` method that accepts a `KeyValueMapper` parameter that you use to select a new key. We'll see an example of selecting a new key in the next example.

7.2.1 Group By details

We should take a quick detour to briefly discuss the return type of the group-by call, which is a `KGroupedStream`. The `KGroupedStream` is an intermediate object and it provides methods `aggregate`, `count`, and `reduce`. In most cases, you won't need to keep a reference to the `KGroupedStream`, you'll simply execute the method you need and its existence is transparent to you.

What are the cases when you'd want to keep a reference to the `KGroupedStream`? Any time you want to perform multiple aggregation operations from the same key grouping is a good example. We'll see one when we cover windowing later on. Now let's get back to the discussion of our first stateful operation.

Immediately after the `groupByKey` call we execute `reduce`, and as I've explained before the `KGroupedStream` object is transparent to us in this case. The `reduce` method has overloads taking anywhere from one to three parameters, in this case we're using the two parameter version which accepts a `Reducer` interface and a `Materialized` configuration object as parameters. For the `Reducer` you're using a method reference to the static method `Double.sum` which sums the previous total score with the newest score from the game.

The `Materialized` object provides the serdes used by the state store for (de)serializing keys and values. Under the covers, Kafka Streams uses local storage to support stateful operations. The stores store key-value pairs as byte arrays, so you need to provide the serdes to serialize records on input and deserialize them on retrieval. We'll get into the details of state stores in an upcoming section.

After `reduce` you call `toStream` because the result of all aggregation operations in Kafka Streams is a `KTable` object (which we haven't covered yet, but we will in the next chapter), and to forward the aggregation results to downstream operators we need to convert it to a `KStream`.

Then we can send the aggregation results to an output topic via a sink node represented by the `to` operator. But stateful processors don't have the same forwarding behavior as stateless ones, so we'll take a minute here to describe that difference.

Kafka Streams provides a caching mechanism for the results of stateful operations. Only when Kafka Streams flushes the cache are stateful results forwarded to downstream nodes in the topology. There are two scenarios when Kafka Streams will flush the cache. The first when the cache is full, which by default is 10MB, or secondly when Kafka Streams commits, which is every thirty seconds with default settings. An illustration of this will help to cement your understanding of how the caching works in Kafka Streams.

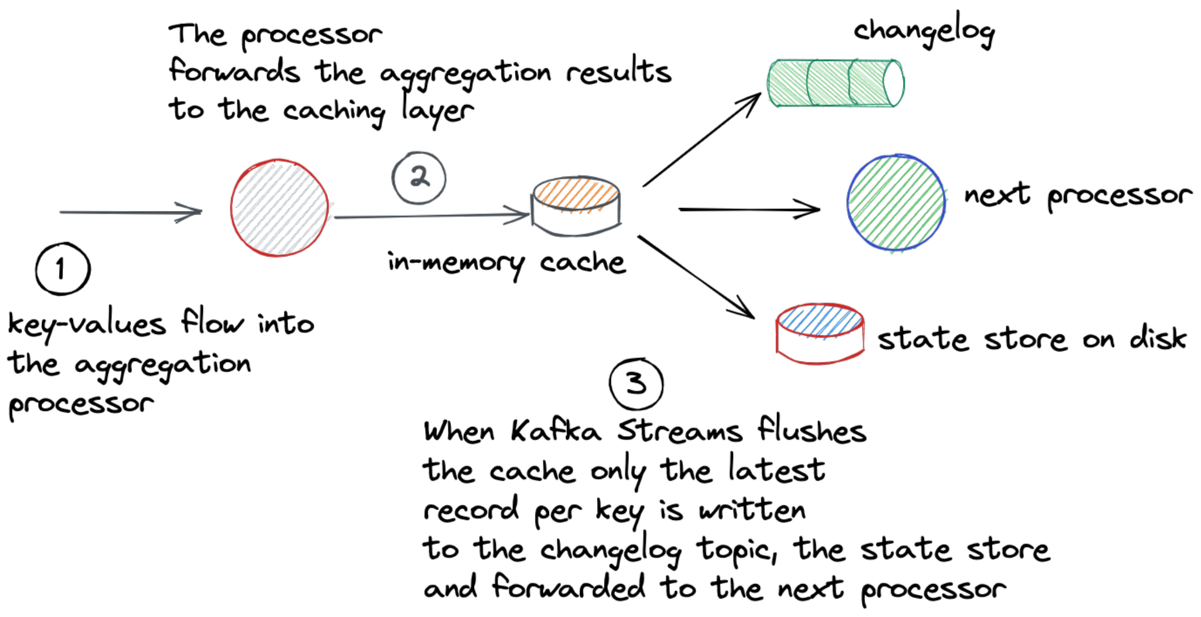
The processor
forwards the aggregation results
to the caching layer
changelog
2
next processor
1
in-memory cache
key-values flow into
the aggregation
processor
state store on disk
3
When Kafka Streams flushes
the cache only the latest
record per key is written
to the changelog topic, the state store
and forwarded to the next processor

Figure 7.3 Caching intermediate results of an aggregation operation

So from looking at the illustration you can see that the cache sits in front forwarding records and as a result you won't observe several of the intermediate results, but you will always see the latest updates at the time of a cache flush. This also has the effect of limiting writes to the state store and its associated changelog topic. Changelog topics are internal topics created by Kafka Streams for fault tolerance of the state stores. We'll cover changelog topics in an upcoming section.

TIP **If you want to observe every result of a stateful operation you can disable the cache by setting the** `StreamsConfig.CACHE_MAX_BYTES_BUFFERING_CONFIG` **setting to 0.**

7.2.2 Aggregation vs. reducing

At this point you've learned about one stateful operator, but we have another option for stateful operations. If you noticed with `reduce`, since you are merging record values, it's expected that a `reduce` returns the same type as a result. But sometimes you'll want to build a different result type and for that you'll want to use the `aggregate` operation. The concept behind an aggregation is similar, but you have the flexibility to return a type different from the record value. Let's look at an example to answer why you would use `aggregate` over `reduce`.

Imagine you work for ETrade you need to create an application that tracks stock transactions of individual customers, not large institutional traders. You want to keep a running tally of the total volume of shares bought and sold, the dollar volume of sales and purchases, and the highest and lowest price seen at any point.

To provide this type of information, you'll need to create a custom data object. This where the `aggregate` comes into play, because it allows for a different return type from the incoming value. In this case the incoming record type is singular stock transaction object, but the aggregation result will be a different type containing the required information listed in the previous paragraph.

Since we'll need to put this custom object in a state store which requires serialization, we'll create a Protobuf schema so we can generate it and leverage utility methods for creating Protobuf serdes . Since this application has detailed aggregation requirements, we'll implement the `Aggregator<K, V, VR>` interface as a concrete class which will allow us to test it independently.

Let's take a look at part of the aggregator implementation. Since this class contains some logic not directly related to learning Kafka Streams, I'm only going to show partial information on the class, to view full details, consult the source code and look for the `bbejeck.chapter_7.aggregator.StockAggregator` class.

Listing 7.4 Aggregator implementation used for creating stock transaction summaries

```
public class StockAggregator implements Aggregator<String,
                                             Transaction,
                                             Aggregate> {

    @Override
    public Aggregate apply(String key,
                          Transaction transaction,
                          Aggregate aggregate) {   ❶

   Aggregate.Builder currAggregate =
                                aggregate.toBuilder();   ❷

    double transactionDollars =
           transaction.getNumberShares()
         * transaction.getSharePrice();   ❸

    if (transaction.getIsPurchase()) {        ❹
        long currentPurchaseVolume =
             currAggregate.getPurchaseShareVolume();
        currAggregate.setPurchaseShareVolume(
                      currentPurchaseVolume
                     + transaction.getNumberShares());

        double currentPurchaseDollars =
                currAggregate.getPurchaseDollarAmount();

        currAggregate.setPurchaseDollarAmount(
                      currentPurchaseDollars
                      + transactionDollars);
    }
    //Further details left out for clarity
```

1. Implementation of the apply method the second parameter is the incoming record, third parameter is the current aggregate
2. Need to use a builder to update Protobuf object
3. Getting the total dollars of the transaction
4. If the transaction is a purchase update the purchase related details

I'm not going to go into much detail about the `Aggregator` instance here, since the main point of this section how to build a Kafka Streams aggregation application, the particulars of how you implement the aggregation is going to vary from case to case. But from looking at this code, you can see how we're building up the transactional data for a given stock. Now let's look at how we'll plug this `Aggregator` implementation into a Kafka Streams application to capture the information. The source code for this example can be found in bbejeck.chapter_7.StreamsStockTransactionAggregations

NOTE There's some details I'm going to leave out of the source code as presented in the book, printing records to the console for example. Going forward our Kafka Streams applications will get more complex and it will be easier to learn the essential part of the lesson if I only show the key details. Rest assured the source code is complete and will be thoroughly tested to ensure that the examples compile and run.

Listing 7.5 Kafka Streams aggregation

```
 KStream<String, Transaction> transactionKStream =
    builder.stream("stock-transactions",
                   Consumed.with(stringSerde, txnSerde));   ❶


transactionKStream.groupBy((key, value) -> value.getSymbol(),   ❷
      Grouped.with(Serdes.String(), txnSerde))
  .aggregate(() -> initialAggregate,   ❸
            new StockAggregator(),
            Materialized.with(stringSerde, aggregateSerde))
  .toStream()  ❹
  .peek((key, value) -> LOG.info("Aggregation result {}", value))
  .to("stock-aggregations", Produced.with(stringSerde, aggregateSerde));   ❺
```

1. Creating the `KStream` instance
2. Grouping by key and providing a function to select the key
3. Calling the aggregate function
4. Converting the resulting aggregation `KTable` to a `KStream`
5. Writing the aggregation results out to a topic

In annotation one, this application starts out in familiar territory, that is creating the `KStream` instance by subscribing it to a topic and providing the serdes for deserialization. I want to call your attention to annotation two, as this is something new.

You've seen a group-by call in the reduce example, but in this example the inbound records are keyed by the client-id and we need to group records by the stock ticker or symbol. So to accomplish the key change, you use `GroupBy` which takes a `KeyValueMapper`, which is a lambda function in our code example. In this case the lambda returns the ticker symbol in the record to enable to proper grouping.

Since the topology changes the key, Kafka Streams needs to repartition the data. I'll discuss repartitioning in more detail in the next section, but for now it's enough to know that Kafka Streams takes care of it for you.

Listing 7.6 Kafka Streams aggregation

```
transactionKStream.groupBy((key, value) -> value.getSymbol(),
      Grouped.with(Serdes.String(), txnSerde))
  .aggregate(() -> initialAggregate,    ❶
            new StockAggregator(),
            Materialized.with(stringSerde, aggregateSerde))
  .toStream()   ❷
  .peek((key, value) -> LOG.info("Aggregation result {}", value))
  .to("stock-aggregations", Produced.with(stringSerde, aggregateSerde));   ❸
```

❶ Calling the aggregate function

❷ Converting the resulting aggregation `KTable` to a `KStream`

❸ Writing the aggregation results out to a topic

At annotation three is where we get to the crux of our example, applying the aggregation operation. Aggregations are little different from the reduce operation in that you need to supply an initial value.

Providing an initial value is required, because you need an existing value to apply for the first aggregation as the result could possibly be a new type. With the reduce, if there's no existing value, it simply uses the first one it encounters.

Since there's no way for Kafka Streams to know what the aggregation will create, you need to give it an initial value to seed it. In our case here it's an instantiated `StockAggregateProto.Aggregate` object, with all the fields uninitialized.

The second parameter you provide is the `Aggregator` implementation, which contains your logic to build up the aggregation as it is applied to each record it encounters. The third parameter, which is optional, is a `Materialized` object which you're using here to supply the serdes required by the state store.

The final parts of the application are used to covert the `KTable` resulting from the aggregation to a `KStream` so that you can forward the aggregation results to a topic. Here you're also using a `peek` operate before the sink processor to view results without consuming from a topic. Using a `peek` operator this way is typically for development or debugging purposes only.

NOTE **Remember when running the examples that Kafka Streams uses caching so you won't immediately observe results until the cache gets flushed either because it's full or Kafka Streams executes a commit.**

So at this point you've learned about the primary tools for stateful operations in the Kafka Streams DSL, reduce and aggregation. There's another stateful operation that deserves mention here and that is the `count` operation. The count operation is a convenience method for a incrementing counter aggregation. You'd use the `count` when you simply need a running tally of a total, say the number of times a user has logged into your site or the total number of readings from an IoT sensor. We won't show an example here, but you can see one in action in the source code at bbejeck/chapter_7/StreamsCountingApplication.

In this previous example here where we built stock transaction aggregates, I mentioned that changing the key for an aggregation requires a repartitioning of the data, let's discuss this in a little more detail in the next section.

7.2.3 Repartitioning the data

In the aggregation example we saw how changing the key required a repartition. Let's have a more detailed conversation on why Kafka Streams repartition and how it works. Let's talk about the why first.

We learned in a previous chapter that the key of a Kafka record determines the partition. When you modify or change the key, there's a strong probability it belongs on another partition. So, if you've changed the key and you have a processor that depends on it, an aggregation for example, Kafka Streams will repartition the data so the records with the new key end up on the correct partition. Let's look at an illustration demonstrating this process in action:

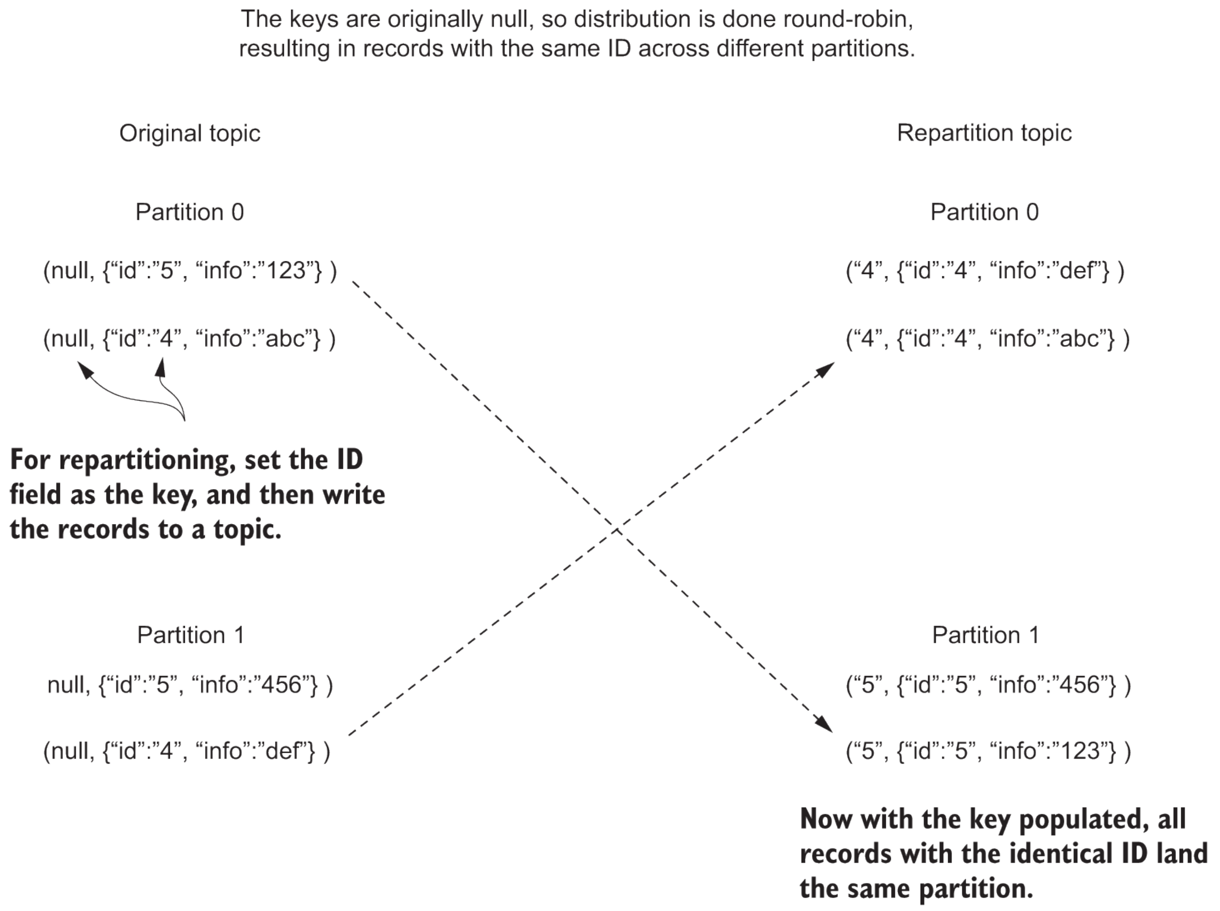
The keys are originally null, so distribution is done round-robin, resulting in records with the same ID across different partitions.
Original topic
Partition 0
(null, {“id”:”5”, “info”:”123”})
(null, {“id”:”4”, “info”:”abc”})
For repartitioning, set the ID field as the key, and then write the records to a topic.
Partition 1
null, {“id”:”5”, “info”:”456”})
(null, {“id”:”4”, “info”:”def”})
Repartition topic
Partition 0
(“4”, {“id”:”4”, “info”:”def”})
(“4”, {“id”:”4”, “info”:”abc”})
Partition 1
(“5”, {“id”:”5”, “info”:”456”})
(“5”, {“id”:”5”, “info”:”123”})
Now with the key populated, all records with the identical ID land the same partition.

Figure 7.4 Repartitioning: changing the original key to move records to a different partition

As you can see here, repartitioning is nothing more than producing records out to a topic and then immediately consuming them again. When the Kafka Streams embedded producer writes the records to the broker, it uses the updated key to select the new partition. Under the covers, Kafka Streams inserts a new sink node for producing the records and a new source node for consuming them, here’s an illustration showing the before and after state where Kafka Streams updated the topology:

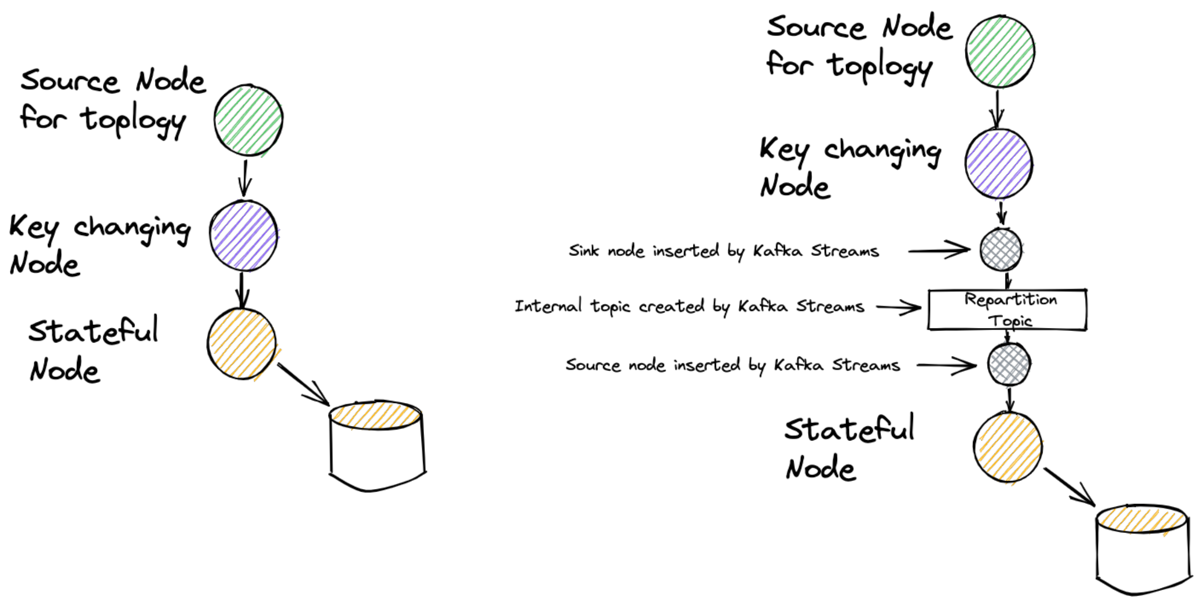

Figure 7.5 Updated topology where Kafka Streams adds a sink and source node for repartitioning of the data

The newly added source node creates a new sub-topology in the overall topology for the application. A sub-topology is a portion of a topology that share a common source node. Here's an updated version of the repartitioned topology demonstrating the sub-topology structures:

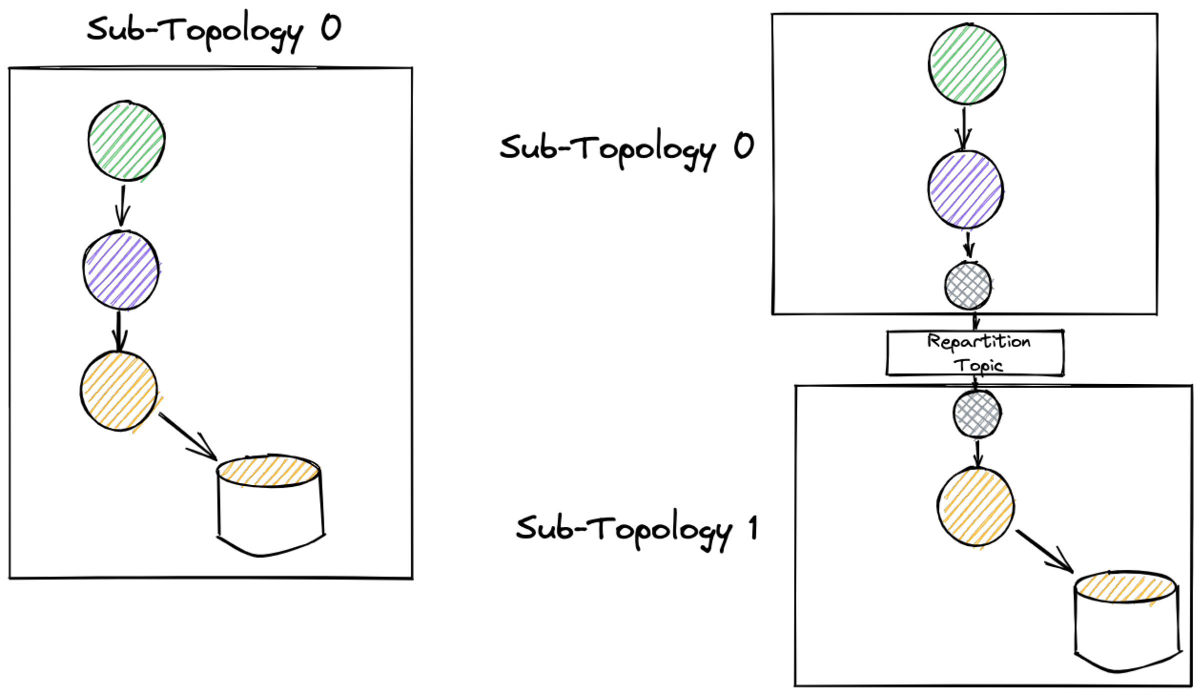

Figure 7.6 Adding a sink and source node for repartitioning creates a new sub-topology

So any processors that come after the new source node are part of the new sub-topology.

What is the determining factor that causes Kafka Streams to repartition? If you have a key

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/kafka-streams-in-action-second-edition/discussion

changing operation ***and*** there's a downstream operation that relies on the key, such as a `groupByKey`, aggregation, or join (we'll get to joins soon). Otherwise, if there are no downstream operations dependent on the key, Kafka Streams will leave the topology as is. Let's look a couple of examples to help clarify this point:

Listing 7.7 Examples of when repartitioning is needed

```
myStream.groupBy(...).reduce(...)...   ❶

myStream.map(...).groupByKey().reduce(...)...   ❷

filteredStream = myStream.selectKey(...).filter(...);   ❸
....
filteredStreaam.groupByKey().aggregate(...)...   ❸
```

❶ Using `groupBy` followed by a `reduce`

❷ Executing a `map` followed by a `groupByKey`

❸ Using a `selectKey` to choose a new key and the resulting `KStream` later calls `groupByKey`

What these code examples demonstrate is when you execute an operation where you ***could*** change the key, Kafka Streams sets an internal boolean flag, `repartitionRequired`, to `true`. Since Kafka Streams can't possibly know if you changed the key or not, when it finds an operation dependent on the key and the internal flag evaluates to `true`, it will automatically repartition the data.

On the other hand, even if you change the key, but don't do an aggregation or join, the topology remains the same:

Listing 7.8 Examples of when repartitioning is not needed

```
myStream.map(...).peek(...).to(...);   ❶

myStream.selectKey(...).filter(...).to(...);   ❷
```

❶ Using a map but no downstream operation depends on the key

❷ Using a selectKey but also no downstream operations rely on the key

In these examples, even if you updated the key, it doesn't affect the results of the downstream operators. For example filtering a record solely depends on if the predicate evaluates to `true` or not. Additionally, since these `KStream` instances write out to a topic, the records with updated keys will end up on the correct partition.

So the bottom line is to only use key-changing operations (`map`, `flatMap`, `transform`) when you actually need to change the key. Otherwise it's best to use processors that only work on values i.e. `mapValues`, `flatMapValues` etc. this way Kafka Streams won't needlessly repartition the

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/kafka-streams-in-action-second-edition/discussion

data. There are overloads to XXXValues methods that provide **access** to the key when updating a value, but changing the key in this case will lead to undefined behavior.

NOTE **The same is true when grouping records before an aggregation. Only use** `groupBy` **when you need to change the key, otherwise favor** `groupByKey`**.**

It’s not that you should avoid repartitioning, but since it adds processing overhead it is a good idea to only do it when required.

Before we wrap up coverage of repartitioning we should talk about an important additional subject; inadvertently creating redundant repartition nodes and ways to prevent it. Let’s say you have an application with two input streams. You need to perform an aggregation on the first stream as well as join it with the second stream. Your code would look something like this:

Listing 7.9 Changing the key then aggregate and join

```
// Several details omitted for clarity

KStream<String, String> originalStreamOne = builder.stream(...);

KStream<String, String> inputStreamOne = originalStreamOne.selectKey(...);   ❶

KStream<String, String> inputStreamTwo = builder.stream(...);   ❷

inputStreamOne.groupByKey().count().toStream().to(...);   ❸

KStream<String, String> joinedStream =         ❹
  inputStreamTwo.join(inputStreamOne,
              (v1, v2)-> v1+":"+v2,
              JoinWindows.ofTimeDifferenceWithNoGrace(...),
              StreamJoined.with(...);

....
```

❶ Changing the key of the original stream setting the "needsRepartition" flag

❷ The second stream

❸ Performing a group-by-key triggering a repartition

❹ Performing a join between inputStreamOne and inputStreamTwo triggering another repartition

This code example here is simple enough. You take the `originalStreamOne` and need you to change the key since you’ll need to do an aggregation and a join with it. So you use a `selectKey` operation, which sets the `repartitionRequired` flag for the returned `KStream`. Then you perform a `count()` and then a `join` with `inputStreamOne`. What is not obvious here is that Kafka Streams will automatically create two repartition topics, one for the `groupByKey` operator and the other for the `join`, but in reality you only need one repartition.

It will help to fully understand what’s going on here by looking at the topology for this example.

Notice there are two reparititions, but you only need the first one where the key is changed.

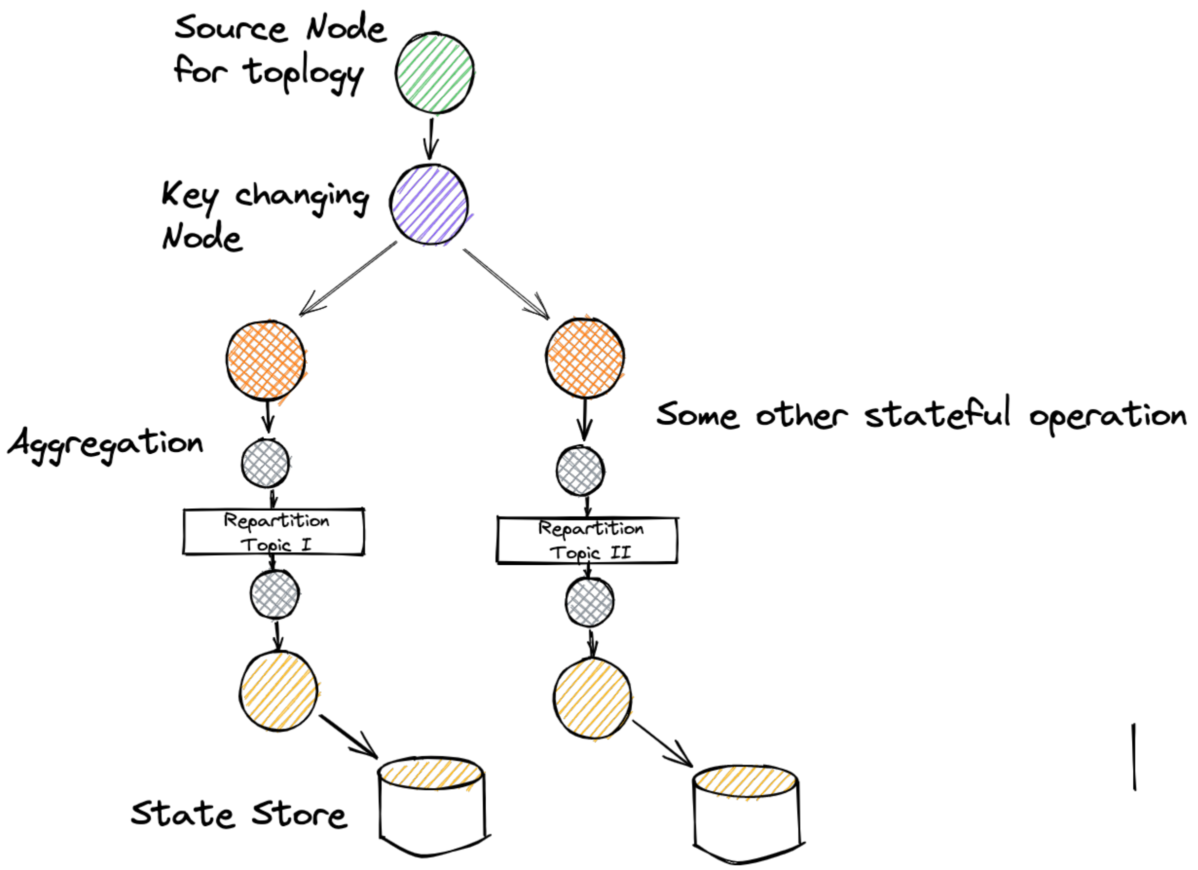

Figure 7.7 Redundant repartition nodes due to a key changing operation occurring previously in the topology

When you used the key-changing operation on `originalStreamOne` the resulting `KStream`, `inputStreamOne`, now carries the `repartitionRequired = true` setting. So any `KStream` resulting from `inputStreamOne` that uses a processor involving the key will trigger a repartition.

What can you do to prevent this from happening? There are two choices here; manually repartition earlier which sets the repartition flag to `false`, so any subsequent streams won't trigger a repartition. The other option is let Kafka Streams handle it for you by enabling optimizations. Let's talk about using the manual approach first.

NOTE **While repartition topics do take up disk space, Kafka Streams actively purges records from them, so you don't need to be concerned with the size on disk, but avoiding redundant repartitions is still a good idea.**

7.2.4 Proactive Repartitioning

For the times when you might need to repartition the data yourself, the KStream API provides the `repartition` method. Here's how you use it to manually repartition after a key change:

Listing 7.10 Changing the key, repartitioning then performing an aggregation and a join

```
//Details left out of this example for clarity

KStream<String, String> originalStreamOne = builder.stream(...);
KStream<String, String> inputStreamOne = originalStreamOne.selectKey(...);   ❶

KStream<String, String> inputStreamTwo = builder.stream(...);

KStream<String, String> repartitioned =
  inputStreamOne.repartition(Repartitioned     ❷
                .with(stringSerde, stringSerde)
                .withName("proactive-repartition"));

repartitioned.groupByKey().count().toStream().to(...);   ❸

KStream<String, String> joinedStream = inputStreamTwo.join(...)   ❹

.....
```

❶ Changing the key setting the "needs repartition" flag

❷ Calling the repartition method and providing key-value serdes and a name for the repartition topic

❸ Performing an aggregation on the repartitioned stream

❹ Performing a join with the repartitioned stream.

The code here has only one change, adding `repartition` operation before performing the `groupByKey`. What happens as a result is Kafka Streams creates a new sink-source node combination that results in a new subtopology. Let’s take a look at the topology now and you’ll see the difference compared to the previous one:

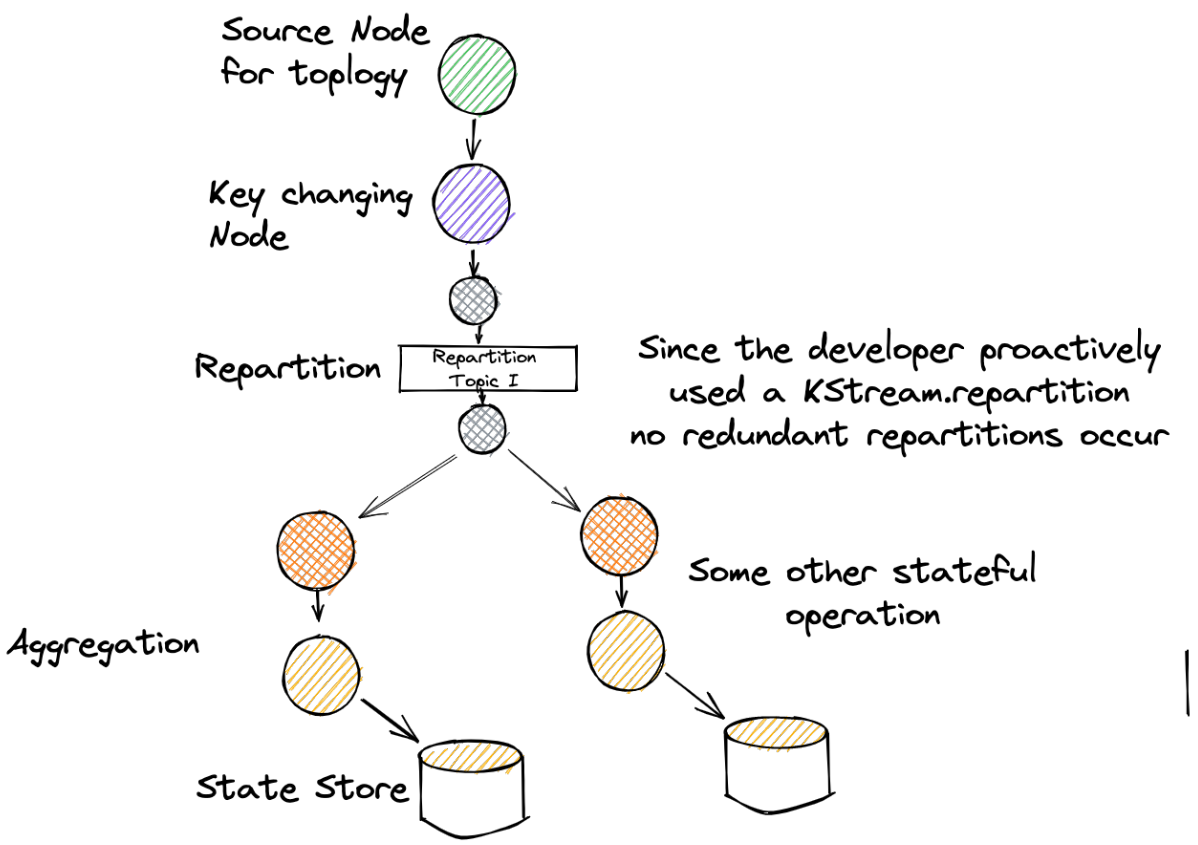
Source Node for toplogy
Key changing Node
Repartition
Repartition Topic I
Since the developer proactively used a KStream.repartition no redundant repartitions occur
Some other stateful operation
Aggregation
State Store

Figure 7.8 Now only one repartition node due to a proactive repartition also allows for more stateful operations without repartitioning

This new sub-topology ensures that the new keys end up on the correct partition, and equally as important, the returned `KStream` object has the `needsRepartition` flag set to `false`. As a result, all downstream stateful operations that are descendants of this `KStream` object don't trigger any further repartitions (unless the one of them changes the key again).

The `KStream.repartition` method accepts one parameter, the `Repartitioned` configuration object. `Repartitioned` allows you to specify:

1. The serdes for the key and value
2. The base name for the topic
3. The number of partitions to use for the topic
4. A `StreamPartitioner` instance should you need customize the distribution of records to partitions

Let's pause on our current discussion and review some of these options. Since I've already covered serdes and the `StreamPartitioner` in the previous chapter, I'm going to leave them out here.

Providing a base-name for the repartition topic is always a good idea. I'm using the term base-name because Kafka Streams takes the name you provide and adds a prefix of "<application-id>-" which comes from the value you supplied in the configs and a suffix of

"-repartition".

So given an application-id of "streams-financial" and a name of "stock-aggregation" results in a repartition topic named "streams-financial-stock-aggregation-repartition". The reason it’s a good idea to always provide a name is two fold. First having a meaningful topic name is always helpful to understand its role when you list the topics on your Kafka cluster.

Secondly, and probably more important, is the name you provide remains fixed even if you change you topology upstream of the repartition. Remember, when you don’t provide names for processors, Kafka Streams generates names for them, and part of the name includes a zero padded number generated by a global counter.

So if you add or remove operators upstream of your repartition operation and you ***haven’t*** explicitly named it, its name will change due to changes in global counter. This name shift can be problematic when re-deploying an existing application. I’ll talk more about importance of naming stateful components of a Kafka Streams application in an upcoming section.

NOTE **Although there are four parameters for the `Repartitioned` object, you don’t have to supply all of them. You can use any combination of the parameters that suit your needs.**

Specifying the number of partitions for the repartition topic is particularly useful in two cases: co-partitioning for joins and increasing the number of tasks to enable higher throughput. Let’s discuss the co-partitioning requirement first. When performing joins, both sides must have the same number of partitions (we’ll discuss why this is so in the upcoming joins section). So by using the `repartition` operation, you can change the number partitions to enable a join, without needing to change the original source topic, keeping the changes internal to the application.

7.2.5 Repartitioning to increase the number of tasks

If you recall from the previous chapter, the number of partitions drive the number of tasks which ultimately determines the amount of active threads a application can have. So one way to increase the processing power is to increase the number of partitions, since that leads to more tasks and ultimate more threads that can process records. Keep in mind that tasks are evenly assigned to all applications with the same id, so this approach to increase throughput is particularly useful in an environment where you can elastically expand the number of running instances.

While you could increase the number of partitions for the source topic, this action might not always be possible. The source topic(s) of a Kafka Streams application are typically "public" meaning other developers and applications use that topic and in most organizations, changes to shared infrastructure resources can be difficult to get done.

Let’s look at an example of performing a repartition to increase the number of tasks (example found in bbejeck.chapter_7.RepartitionForThroughput)

Listing 7.11 Increasing the number of partitions for a higher task count

```
KStream<String, String> repartitioned =

initialStream.repartition(Repartitioned
            .with(stringSerde, stringSerde)
            .withName("multiple-aggregation")
            .withNumberOfPartitions(10));   ❶
```

❶ Increasing the number of partitions

Now this application will have 10 tasks which means there can up to 10 stream threads processing records driven by the increase in the number of partitions.

You need to keep in mind however, that adding partitions for increased throughput will work best when there is a fairly even distribution of keys. For example, if seventy percent of you key space lands on one partition, increasing the number of partitions will only move those keys to a new partition. But since the overall **distribution** of the keys is relatively unchanged, you won’t see any gains in throughput, since one partition, hence one task, is shouldering most of the processing burden.

So far we’ve covered how you can proactively repartition when you’ve changed the key. But this requires you to know when to repartition and always remember to do so, but is there a better approach, maybe have Kafka Streams take care of this for you automatically? Well there is a way, by enabling optimizations.

7.2.6 Using Kafka Streams Optimizations

While you’re busy creating a topology with various methods, Kafka Streams builds a graph or internal representation of it under the covers. You can also consider the graph to be a "logical representation" of your Kafka Streams application. In your code, when you execute `StreamBuilder#build` method, Kafka Streams traverses the graph and builds the final or physical representation of the application.

At a high level, it works like this: as you apply each method, Kafka Streams adds a "node" to the graph as depicted in the following illustration:

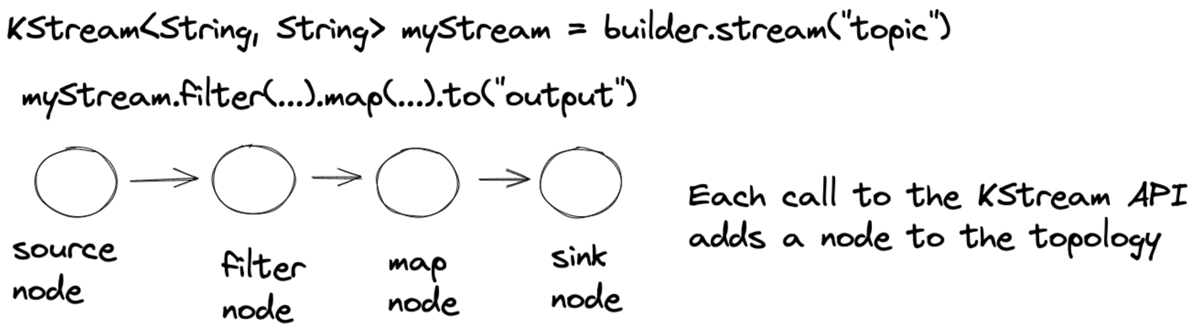
KStream<String, String> myStream = builder.stream("topic")
myStream.filter(...).map(...).to("output")
source node
filter node
map node
sink node
Each call to the KStream API adds a node to the topology

Figure 7.9 Each call to a KStream method adds a node to the graph

When you make an additional method call, the previous "node" becomes the parent of the current one. This process continues until you finish building your application.

Along the way, Kafka Streams will record metadata about the graph it's building, specifically it records if it has encountered a repartition node. Then when use the `StreamsBuilder#build` method to create the final topology, Kafka Streams will examine the graph for redundant repartition nodes, and if found, it will re-write your topology to have only one! This is opt-in behavior for Kafka Streams, so to get this feature working, you'll need to enable optimizations by doing the following:

Listing 7.12 Enabling optimizations in Kafka Streams

```
streamProperties.put(StreamsConfig.TOPOLOGY_OPTIMIZATION_CONFIG,
                         StreamsConfig.OPTIMIZE);     ❶
builder.build(streamProperties);    ❷
```

❶ Enabling optimizations via configuration

❷ Passing properties to the StreamBuilder

So to enable optimizations you need to first set the proper configuration because by default it is turned off. The second step is to pass the properties object to the `StreamBuilder#build` method. Then Kafka Streams will optimize your repartition nodes when building the topology.

NOTE **If you have more than one key-changing operation with a stateful one further downstream the optimizing will not remove that repartition. It only takes away redundant repartitions for single key-changing processor.**

But when you enable optimizations Kafka Streams automatically updates the topology by removing the three repartition nodes preceding the aggregation and inserts a new single repartition node immediately after the key-changing operation which results in a topology that looks like the illustration in the "Proactive Repartitioning" section.

So with a configuration setting and passing the properties to the `StreamBuilder` you can automatically remove any unnecessary repartitions! The decision on which one to use really comes down to personal preference, but by enabling optimizations it guards against you overlooking where you may need it.

Now we’ve covered repartitioning, let’s move on to our next stateful operation, joins

7.3 Stream-Stream Joins

Sometimes you may need to combine records from different event streams to "complete the picture" of what your application is tasked with completing. Say we have stream of purchases with the customer ID as the key and a stream of user clicks and we want to join them so we can make connection between pages visted and purchases. To do this in Kafka Streams you use a join operation. Many of you readers are already familiar with the concept of a join from SQL and the relational database world and the concept is the same in Kafka Streams.

Let’s look at an illustration to demonstrate the concept of joins in Kafka Streams

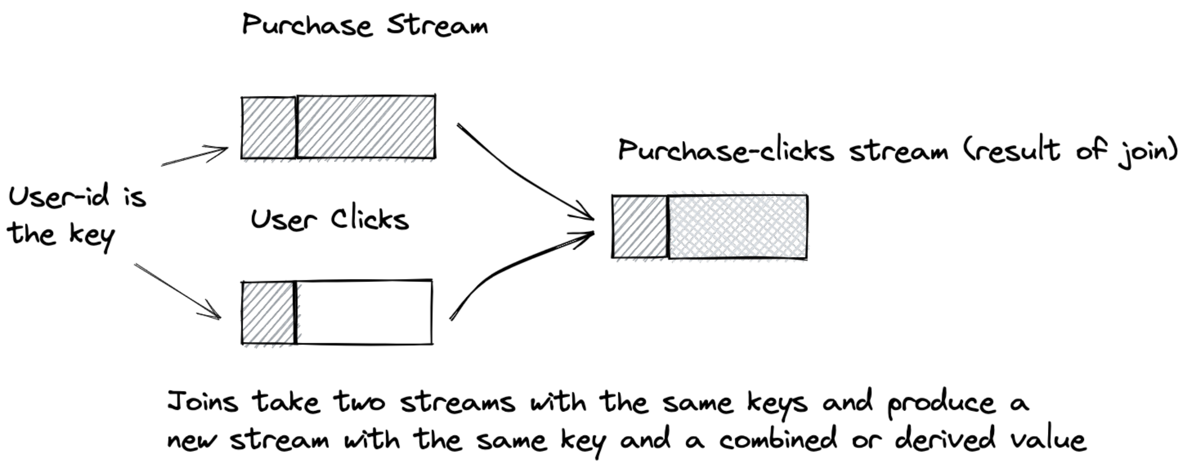
Purchase Stream
User-id is the key
User Clicks
Purchase-clicks stream (result of join)
Joins take two streams with the same keys and produce a new stream with the same key and a combined or derived value

Figure 7.10 Two Streams with the same keys but different values

From looking at the graphic above, there are two event streams that use the same values for the key, a customer id for example, but the values are different. In one stream the values are purchases and the other stream the values are links to pages the user clicked visiting the site.

IMPORTANT **Since joins depend on identical keys from different topics residing on the same partition, the same rules apply when it comes to using a key-changing operation. If a `KStream` instance is flagged with `repartitionRequired`, Kafka Streams will partition it before the join operation. So all the information in the repartitioning section of this chapter applies to joins as well.**

In this section, you’ll take different events from two streams with the same key, and combine them to form a new event. The best way to learn about joining streams is to look at a concrete example, so we'll return to the world of retail. Consider a big box retailer that sells just about anything you can imagine. In an never ending effort to lure more customers in the store, the retailer partners with a national coffee house and it embeds a cafe in each store.

To encourage customers to come into the store, the retailer has started a special promotion where if you are a member of the customer-club and you buy a coffee drink from the embedded cafe and a purchase in the store (in either order), they'll automatically earn loyalty points at the completion of your second purchase. The customers can redeem those points for items from either store. It's been determined that purchases must made within 30 minutes of each other to qualify for the promotion.

Since the main store and the cafe run on separate computing infrastructure, the purchase records are in two event streams, but that's not an issue as they both use the customer id from the club membership for the key, so it's a simply a case of using a stream-stream join to complete the task.

7.3.1 Implementing a stream-stream join

The next step is to perform the actual join. So let's show the code for the join, and since there are a couple of components that make up the join, I'll explain them in a section following the code example. The source code for this example can be found in src/main/java/bbejeck/chapter_7/KafkaStreamsJoinsApp.java).

Listing 7.13 Using the `join()` method to combine two streams into one new stream based on keys of both streams

```
// Details left out for clarity

KStream<String, CoffeePurchase>
                     coffeePurchaseKStream = builder.stream(...)   ❶

KStream<String, RetailPurchase>
                     retailPurchaseKStream = builder.stream(...)   ❶

ValueJoiner<CoffeePurchase,
            RetailPurchase,
            Promotion> purchaseJoiner =
                                      new PromotionJoiner();   ❷

JoinWindows thirtyMinuteWindow =
     JoinWindows.ofTimeDifferenceWithNoGrace(Duration.minutes(30));   ❸

KStream<String, Promotion> joinedKStream =
    coffeePurchaseKStream.join(retailPurchaseKStream,   ❹
                               purchaseJoiner,
                               thirtyMinuteWindow,
                                StreamJoined.with(stringSerde,   ❺
                                                  coffeeSerde,
                                                  storeSerde)
                                      .withName("purchase-join")
                                      .withStoreName("join-stores"));
```

❶ The streams you will join

❷ ValueJoiner instance which produces the joined result object

❸ JoinWindow specifying the max time difference between records to participate in join

❹ Constructs the join

❺ StreamJoined configuration object

You supply four parameters to the `KStream.join` method:

- `retailPurchaseKStream` — The stream of purchases from to join with.
- `purchaseJoiner` — An implementation of the `ValueJoiner<V1, V2, R>` interface. `ValueJoiner` accepts two values (not necessarily of the same type). The `ValueJoiner.apply` method performs the implementation-specific logic and returns a (possibly new) object of type R. In this example, `purchaseJoiner` will add some relevant information from both `Purchase` objects, and it will return a `PromotionProto` object.
- `thirtyMinuteWindow` — A `JoinWindows` instance. The `JoinWindows.ofTimeDifferenceWithNoGrace` method specifies a maximum time difference between the two values to be included in the join. Specifically the timestamp on the secondary stream, `retailPurchaseKStream` can only be a maximum of 30 minutes before or after the timestamp of a record from the `coffeePurchaseKStream` with the same key.
- A `StreamJoined` instance — Provides optional parameters for performing joins. In this case, it's the key and the value `Serde` for the calling stream, and the value `Serde` for the secondary stream. You only have one key `Serde` because, when joining records, keys must be of the same type. The `withName` method provides the name for the node in the topology and the base name for a repartition topic (if required). The `withStoreName` is the base name for the state stores used for the join. I'll cover join state stores usage in an upcoming section.

NOTE `Serde` **objects are required for joins because join participants are materialized in windowed state stores. You provide only one** `Serde` **for the key, because both sides of the join must have a key of the same type.**

Joins in Kafka Streams are one of the most powerful operations you can perform and it's also one the more complex ones to understand. Let's take a minute to dive into the internals of how joins work.

7.3.2 Join internals

Under the covers, the KStream DSL API does a lot of heavy lifting to make joins operational. But it will be helpful for you to understand how joins are done under the covers. For each side of the join, Kafka Streams creates a join processor with its own state store. Here's an illustration showing how this looks conceptually:

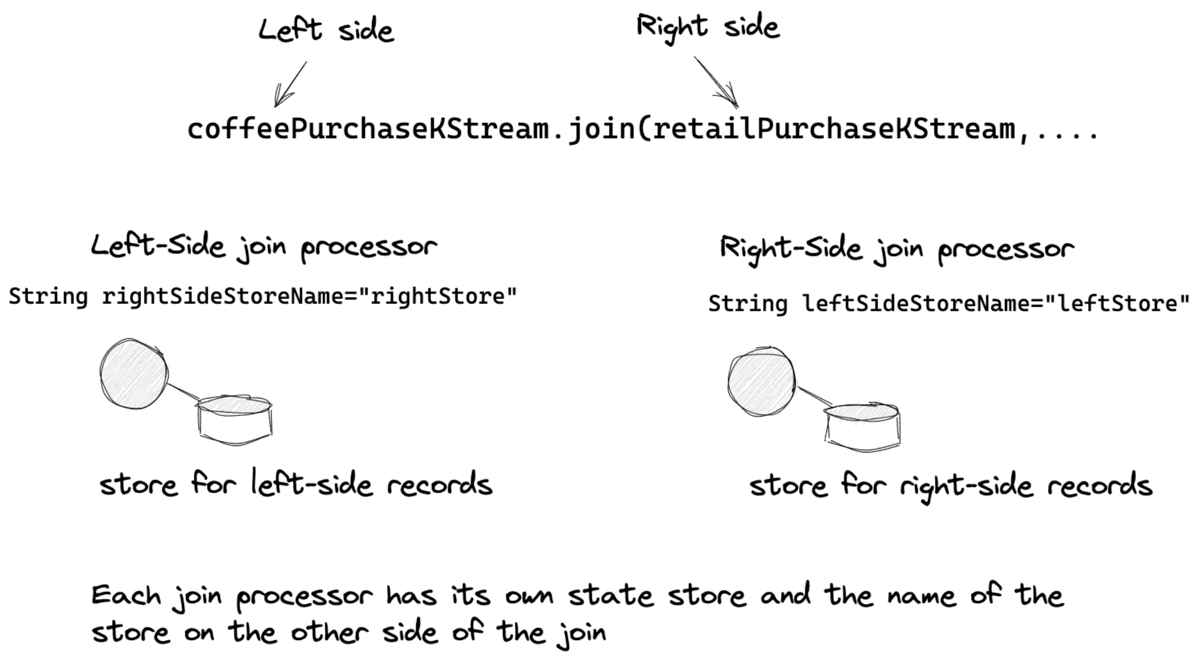
Left side
Right side
coffeePurchaseKStream.join(retailPurchaseKStream,....
Left-Side join processor
String rightSideStoreName="rightStore"
Right-Side join processor
String leftSideStoreName="leftStore"
store for left-side records
store for right-side records
Each join processor has its own state store and the name of the store on the other side of the join

Figure 7.11 In a Stream-Stream join both sides of the join have a processor and state store

When building the processor for the join for each side, Kafka Streams includes the name of the state store for the reciprocal side of the join - the left side gets the name of the right side store and the right side processor contains the left store name. Why does each side contain the name of opposite side store? The answer gets at the heart of how joins work in Kafka Streams. Let's look at another illustration to demonstrate:

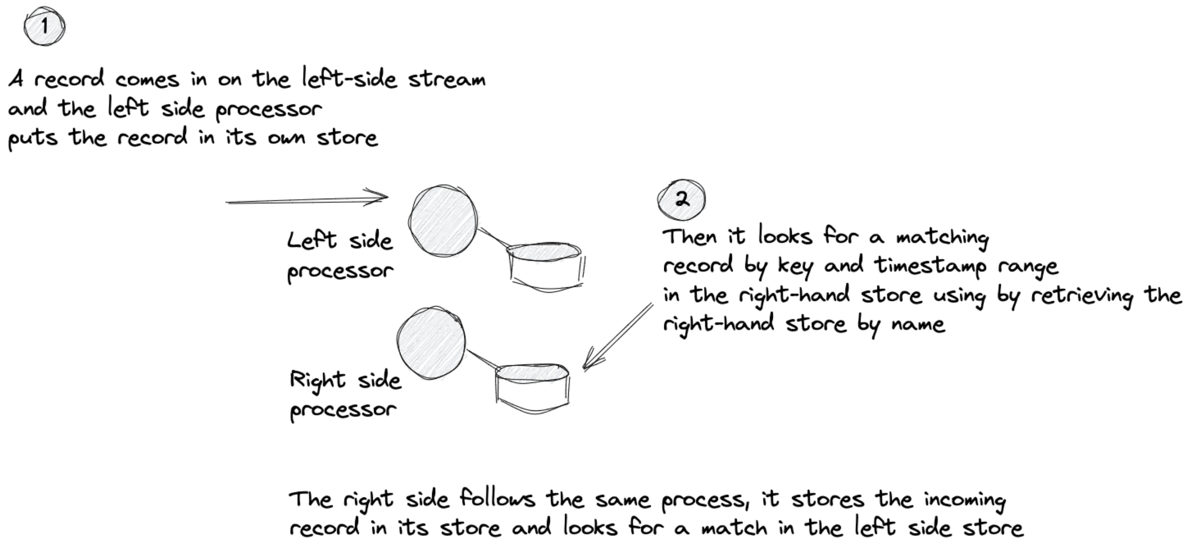
1
A record comes in on the left-side stream and the left side processor puts the record in its own store
Left side processor
Right side processor
2
Then it looks for a matching record by key and timestamp range in the right-hand store using by retrieving the right-hand store by name
The right side follows the same process, it stores the incoming record in its store and looks for a match in the left side store

Figure 7.12 Join processors look in the other side's state store for matches when a new record arrives

When a new record comes in (we're using the left-side processor for the `coffeePurchaseKStream`) the processor puts the record in its own store, but then looks for a match by retrieving the right-side store (for the `retailPurchaseKStream`) by name. The processor retrieves records with the same key and ***within the time range specifed by the JoinWindows***.

Now, the final part to consider is if a match is found. Let's look at one more illustration to help us see what's going on:

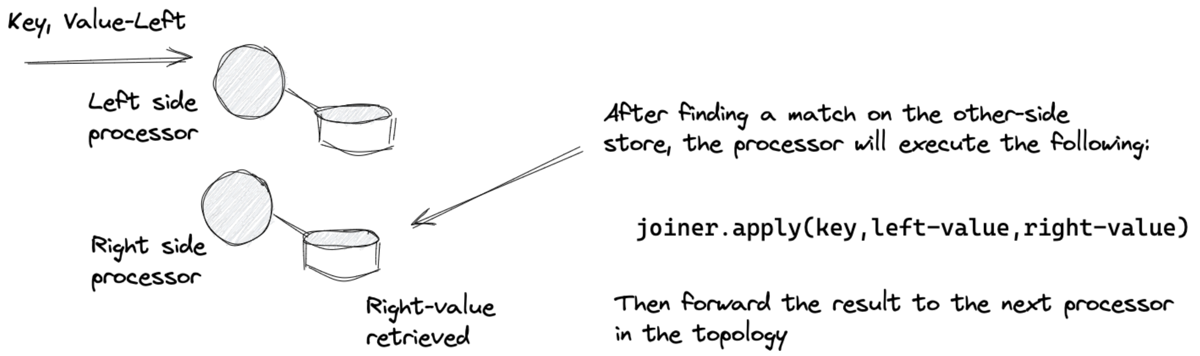
Key, Value-Left
Left side processor
Right side processor
Right-value retrieved
After finding a match on the other-side store, the processor will execute the following:
joiner.apply(key,left-value,right-value)
Then forward the result to the next processor in the topology

Figure 7.13 When matching record(s) is found the processor executes the joiner's apply method with the key, its own record value and the value from the other side

So now, after an incoming record finds a match by looking in the store from the other join side, the join processor (the coffeePurchaseKStream in our illustration) takes the key and the value from its incoming record, the value for each record it has retrieved from the store and executes the `ValueJoiner.apply` method which creates the join record specified by the implementation you've provided. From there the join processor forwards the key and join result to any down stream processors.

Now that we've discussed how joins operate internally let's discuss in more detail some of the parameters to the join

7.3.3 ValueJoiner

To create the joined result, you need to create an instance of a `ValueJoiner<V1, V2, R>`. The `ValueJoiner` takes two objects, which may or may not be of the same type, and it returns a single object, possibly of a third type. In this case, `ValueJoiner` takes a `CoffeePurchase` and a `RetailPurchase` and returns a `Promotion` object. Let's take a look at the code (found in src/main/java/bbejeck/chapter_7/joiner/PromotionJoiner.java).

Listing 7.14 `ValueJoiner` implementation

```
public class PromotionJoiner
     implements ValueJoiner<CoffeePurchase,
                            RetailPurchase,
                            Promotion> {

    @Override
    public Promotion apply(
            CoffeePurchase coffeePurchase,
            RetailPurchase retailPurchase) {

    double coffeeSpend = coffeePurchase.getPrice();    ❶
    double storeSpend = retailPurchase.getPurchasedItemsList()    ❷
            .stream()
            .mapToDouble(pi -> pi.getPrice() * pi.getQuantity()).sum();
    double promotionPoints = coffeeSpend + storeSpend;    ❸
    if (storeSpend > 50.00) {
        promotionPoints += 50.00;
    }
    return Promotion.newBuilder()    ❹
            .setCustomerId(retailPurchase.getCustomerId())
            .setDrink(coffeePurchase.getDrink())
            .setItemsPurchased(retailPurchase.getPurchasedItemsCount())
            .setPoints(promotionPoints).build();
}
```

❶ Extracting how much was spent on coffee

❷ Summing the total of purchased items

❸ Calculating the promotion points

❹ Build and return the new Promotion object

To create the `Promotion` object, you extract the amount spent from both sides of the join and perform a calculation resulting in the total amount of points to reward the customer. I'd like to point out that the `ValueJoiner` interface only has one method, `apply`, so you could use a lambda to represent the joiner. But in this case you create a concrete implementation, because you can write a separate unit test for the `ValueJoiner`. We'll come back this approach in the chapter on testing.

NOTE **Kafka Streams also provides a `ValueJoinerWithKey` interface which provides access to the key for calculating the value of the join result. However the key is considered read-only and making changes to it in the joiner implementation will lead undefined behavior.**

7.3.4 Join Windows

The `JoinWindows` configuration object plays a critical role in the join process; it specifies the difference between the timestamps of records from both streams to produce a join result.

Let's refer to the following illustration as an aid to understand the `JoinWindows` role.

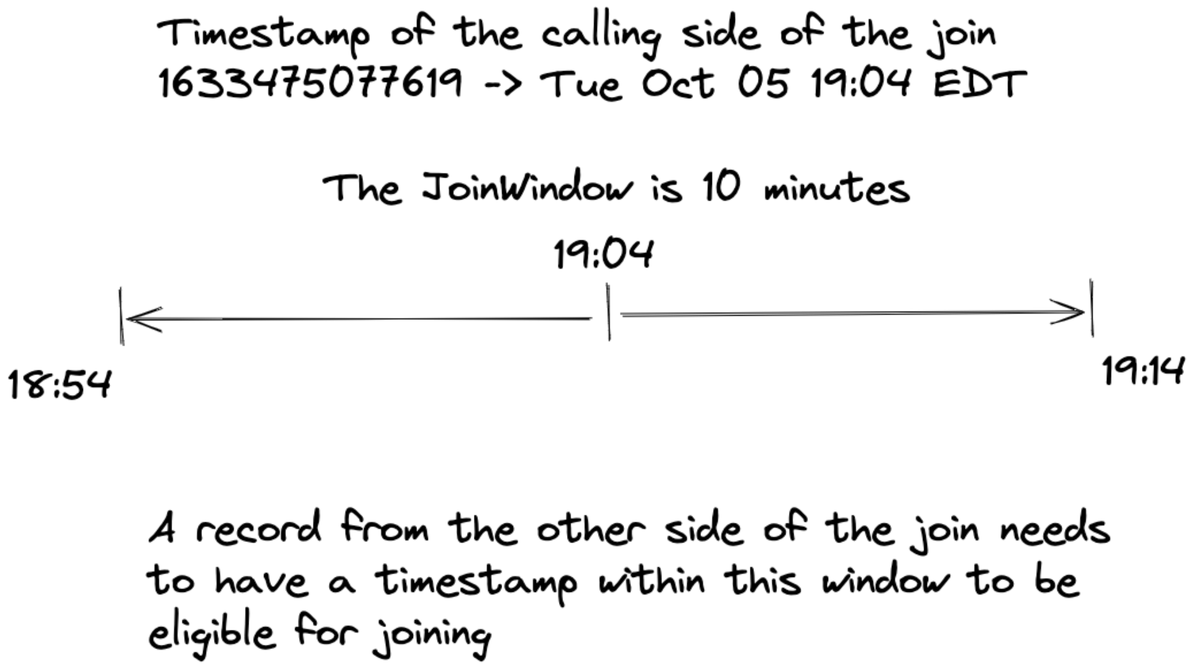
Timestamp of the calling side of the join
1633475077619 -> Tue Oct 05 19:04 EDT
The JoinWindow is 10 minutes
19:04
18:54
19:14
A record from the other side of the join needs to have a timestamp within this window to be eligible for joining

Figure 7.14 The JoinWindows configuration specifies the max difference (before or after) from the timestamp of the calling side the secondary side can have to create a join result.

More precisely the `JoinWindows` setting is the maximum difference, either before or after, the secondary (other) side's timestamp can from the primary side timestamp to create a join result. Looking at the example in listing XXX, the join window there is set for thirty minutes. So let's say a record from the `coffeeStream` has a timestamp of 12:00 PM, for a corresponding record in the `storeStream` to complete the join, it will need a timestamp from 11:30 AM to 12:30 PM.

There are two additional `JoinWindows()` methods are available `after` and `before`, which you can use to specify the timing and possibly the order of events for the join.

Let's say you're fine with the opening window of the join at thirty minutes but you want the closing window to be shorter, say five minutes. You'd use the `JoinWindows.after` method (still using the example in listing XXX) like so

Listing 7.15 Using the JoinWindows.after method to alter the closing side of the join window

```
coffeeStream.join(storeStream,...,
    thirtyMinuteWindow.after(Duration.ofMinutes(5))....
```

Here the opening window stays the same, the `storeStream` record can have a timestamp of at least 11:30 AM , but the closing window is shorter, the latest it can be is now 12:05 PM.

The `JoinWindows.before` method works in a similar manner, just in the opposite direction. Let's say now you want to shorten the opening window, so you'll now use this code:

Listing 7.16 The JoinWindows.before method changes the opening side of the join window

```
coffeeStream.join(storeStream,...,
    thirtyMinuteWindow.before(Duration.ofMinutes(5))....
```

Now you've changed things so the timestamp of the `storeStream` record can be at most 5 minutes *before* the timestamp of a `coffeeStream` record. So the acceptable timestamps for a join (`storeStream` records) now start at 11:55 AM but end at 12:30 PM. You can also use `JoinWindows.before` and `JoinWindows.after` to specify the order of arrival of records to perform a join.

For example to set up a join when a store purchase ***only happens within 30 minutes after a cafe purchase*** you would use `JoinWindows.of(Duration.ofMinutes(0).after(Duration.ofMinutes(30)` and to only consider store purchases ***before*** you would use `JoinWindows.of(Duration.ofMinutes(0).before(Duration.ofMinutes(30))`.

IMPORTANT **In order to perform a join in Kafka Streams, you need to ensure that all join participants are co-partitioned, meaning that they have the same number of partitions and are keyed by the same type. Co-partitioning also requires all Kafka producers to use the same partitioning class when writing to Kafka Streams source topics. Likewise, you need to use the same** `StreamPartitioner` **for any operations writing Kafka Streams sink topics via the** `KStream.to()` **method. If you stick with the default partitioning strategies, you won't need to worry about partitioning strategies.**

As you can see the `JoinWindows` class gives you plenty of options to control joining two streams. It's important to remember that it's the timestamps on the records driving the join behavior. The timestamps can be either the ones set by Kafka (broker or producer) or they can be embedded in the record payload itself. To use a timestamp embedded in the record you'll need to provide a custom `TimestampExtractor` and I'll cover that as well as timestamp semantics in the next chapter.

7.3.5 StreamJoined

The final paramter to discuss is the `StreamJoined` configuration object. With `StreamJoined` you can provide the serdes for the key and the values involved in the join. Providing the serdes for the join records is always a good idea, because you may have different types than what has been configured at the application level. You can also name the join processor and the state stores used for storing record lookups to complete the join. The importance of naming state stores is covered in the upcoming 7.4.5 section.

Before we move on from joins let’s talk about some of the other join options available.

7.3.6 Other join options

The join in listing for the current example is an *inner join*. With an inner join, if either record isn’t present, the join doesn’t occur, and you don’t emit a `Promotion` object. There are other options that don’t require both records. These are useful if you need information even when the desired record for joining isn’t available.

7.3.7 Outer joins

Outer joins always output a record, but the result may not include both sides of the join. You’d use an outer join when you wanted to see a result regardless of a successful join or not. If you wanted to use an outer join for the join example, you’d do so like this:

```
coffeePurchaseKStream.outerJoin(retailPurchaseKStream,..)
```

An outer join sends a result that contains records from either side or both. For example the join result could be `left+right`, `left+null`, or `null+right`, depending on what’s present. The following illustration demonstrates the three possible outcomes of the outer join.

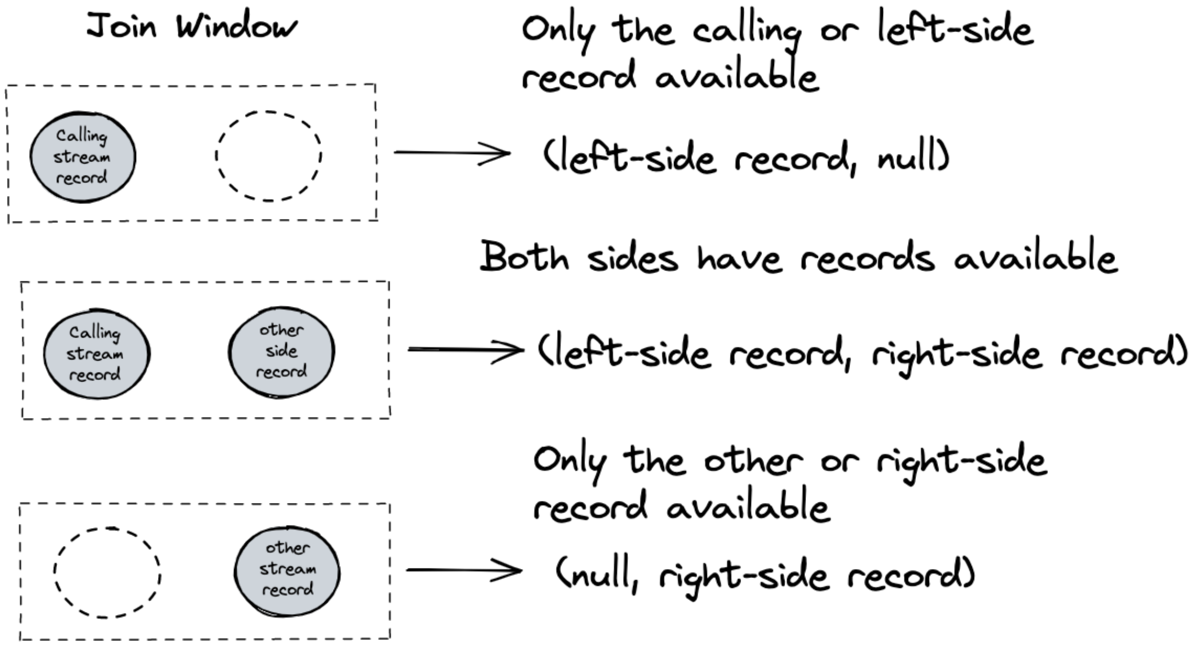
Join Window
Only the calling or left-side record available
Calling stream record
(left-side record, null)
Both sides have records available
Calling stream record
other side record
(left-side record, right-side record)
Only the other or right-side record available
other stream record
(null, right-side record)

Figure 7.15 Three outcomes are possible with outer joins: only the calling stream’s event, both events, and only the other stream’s event.

7.3.8 Left-outer join

A left-outer join also always produces a result. But the difference from the outer-join is the left or calling side of the join is always present in the result, `left+right` or `left+null` for example. You'd use a left-outer join when you consider the left or calling side stream records essential for your business logic. If you wanted to use a left-outer join in listing 7.13, you'd do so like this:

```
coffeePurchaseKStream.leftJoin(retailPurchaseKStream..)
```

Figure 7.17 shows the outcomes of the left-outer join.

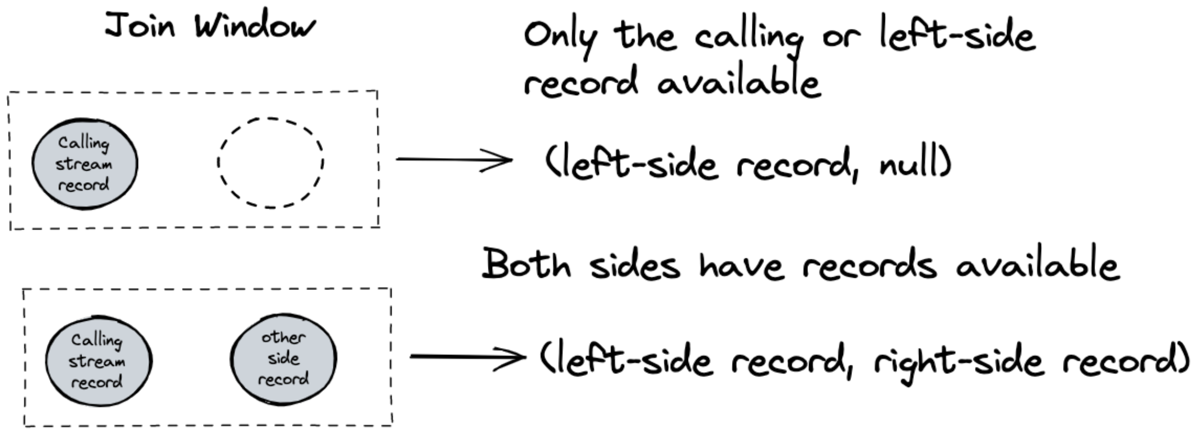
Join Window
Only the calling or left-side record available
Calling stream record
(left-side record, null)
Both sides have records available
Calling stream record
other side record
(left-side record, right-side record)

Figure 7.16 Two outcomes are possible with the left-outer join left and right side or left and null.

At this point you've learned the different join types, so what are the cases when you need to use them? Let's start with the current join example. Since you are determining a promotional reward based on the purchase of two items, each in their own stream an inner-join makes sense. If there is no corresponding purchase on the other side, then you don't have an actionable result, so to emit nothing is desired.

For cases where one side of the join is critical and the other is useful, but not essential then a left-side join is a good choice where you'd use the critical stream on the left or calling side. I'll cover an example when we get to stream-table joins in an upcoming section.

Finally, for a case where you have two streams where both sides enhance each other, but each one is important on its own, then an outer join fits the bill. Consider IoT, where you have two related sensor streams. Combining the sensor information provides you with a more complete picture but you want information from either side if it's available.

In the next section, let's go into the details of the workhorse of stateful operations, the state store.

7.4 State stores in Kafka Streams

So far, we've discussed the stateful operations in the Kafka Streams DSL API, but glossed over the underlying storage mechanism those operations use. In this section, we'll look at the essentials of using state stores in Kafka Streams and the key factors related to using state in streaming applications in general. This will enable you to make practical choices when using state in your Kafka Streams applications.

Before I go into any specifics, let's cover some general information. At a high-level, the state stores in Kafka Streams are key-value stores and they fall into two categories, persistent and in-memory. Both types are durable due to the fact that Kafka Streams uses changelog topics to back the stores. I'll talk more about changelog topics soon.

Persistent stores store their records in local disk, so they maintain their contents over restarts. The in-memory stores place records well, in memory, so they need to be restored after a restart. Any store that needs restoring will use the changelog topic to accomplish this task. But to understand how a state store leverages a changelog topic for restoration, let's take a look at how Kafka Streams implements them.

In the DSL, when you apply a stateful operation to the topology, Kafka Streams creates a state store for the processor (persistent are the default type). Along with the store, Kafka Streams also creates a changelog topic backing the store at the same time. As records are written the store, they are also written to the changelog. Here's an illustration depicting this process:

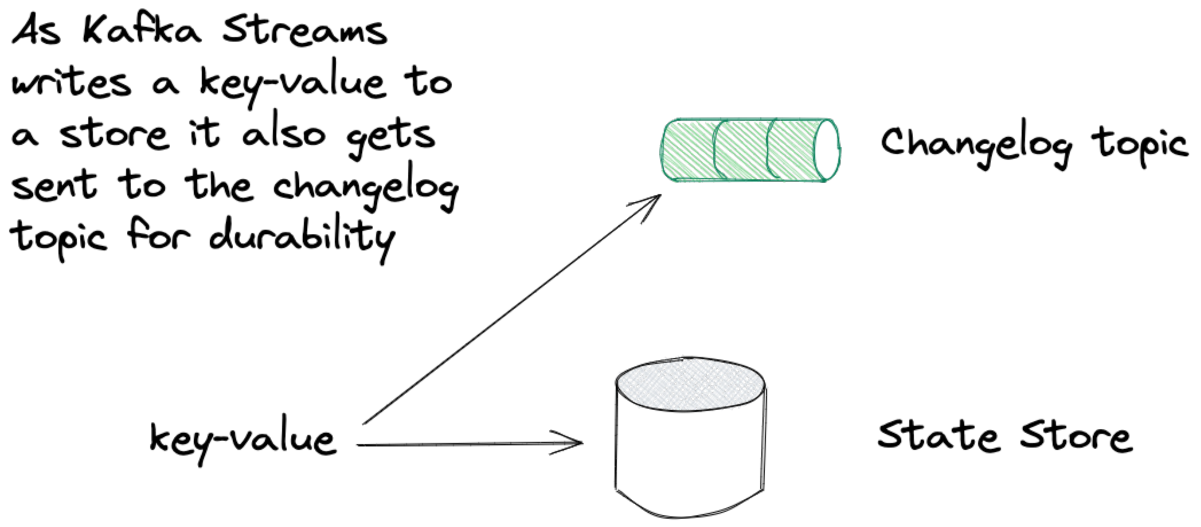
As Kafka Streams writes a key-value to a store it also gets sent to the changelog topic for durability
Changelog topic
key-value
State Store

Figure 7.17 As the key-value records get written to the store they also get written to the changelog topic for data durability

So as Kafka Streams places record into a state store, it also sends it to a Kafka topic that backs the state store. Now if you remember from earlier in the chapter, I mentioned that with an aggregation you don't see every update as Kafka Streams uses a cache to initially hold the results. It’s only on when Kafka Streams flushes the cache, either at a commit or when it’s full, that records from the aggregation go to downstream processors. It’s at this point that Kafka Streams will produce records to the changelog topic.

NOTE **If you’ve disabled the cache then every record gets sent to the state store so this also means every record goes to the changelog topic as well.**

7.4.1 Changelog topics restoring state stores

So how does the Kafka Stream leverage the changelog topic? Let’s first consider the case of a in-memory state store. Since an in-memory store doesn’t maintain it’s contents across restarts, when starting up, any in-memory stores will rebuild their contents from head record of the changelog topic. So even though the in-memory store loses all its contents on application shut-down, it picks up where it left off when restarted.

For persistent stores, usually it’s only after all local state is lost, or if data corruption is detected that it will need to do a ***full*** restore. For persistent stores, Kafka Streams maintains a checkpoint file for persistent stores and it will use the offset in the file as a starting point to restore from instead of restoring from scratch. If the offset is no longer valid, then Kafka Streams will remove the checkpoint file and restore from the beginning of the topic.

This difference in restoration patterns brings an interesting twist to the discussion of the trade-offs of using either persistent or in-memory stores. While an in-memory store should yield faster look-ups as it doesn’t need to go to disk for retrieval, under "happy path" conditions the topology with persistent stores will generally resume to processing faster as it will not have as many records to restore.

IMPORTANT **An exception to using a checkpoint file for restoration is when you run Kafka Streams in EOS mode (either `exactly_once` or `exactly_once_v2` is enabled) as state stores are fully restored on startup to ensure the only records in the stores are ones that were included in successful transactions.**

Another situation to consider is the make up of running Kafka Streams applications. If you recall from our discussion on task assignments, you can change the number of running applications dynamically, either by expansion or contraction. Kafka Streams will automatically assign tasks from existing applications to new members, or add tasks to those still running from an application that has dropped out of the group. A task that is responsible for a stateful operation will have a state store as part of its assignment (I’ll talk about state stores and tasks next).

Let’s consider the case of a Kafka Streams application that loses one of its members, remember you can run Kafka Streams applications on different machines and those with the same application id are considered all part of one logical application. Kafka Streams will issue a

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/kafka-streams-in-action-second-edition/discussion

rebalance and the tasks from the defunct application get reassigned. For any reassigned stateful operations, since Kafka Streams creates a new **empty** store for the newly assigned task, they'll need to restore from the beginning of the changelog topic before they resume processing.

Here's an illustration demonstrating this situation:

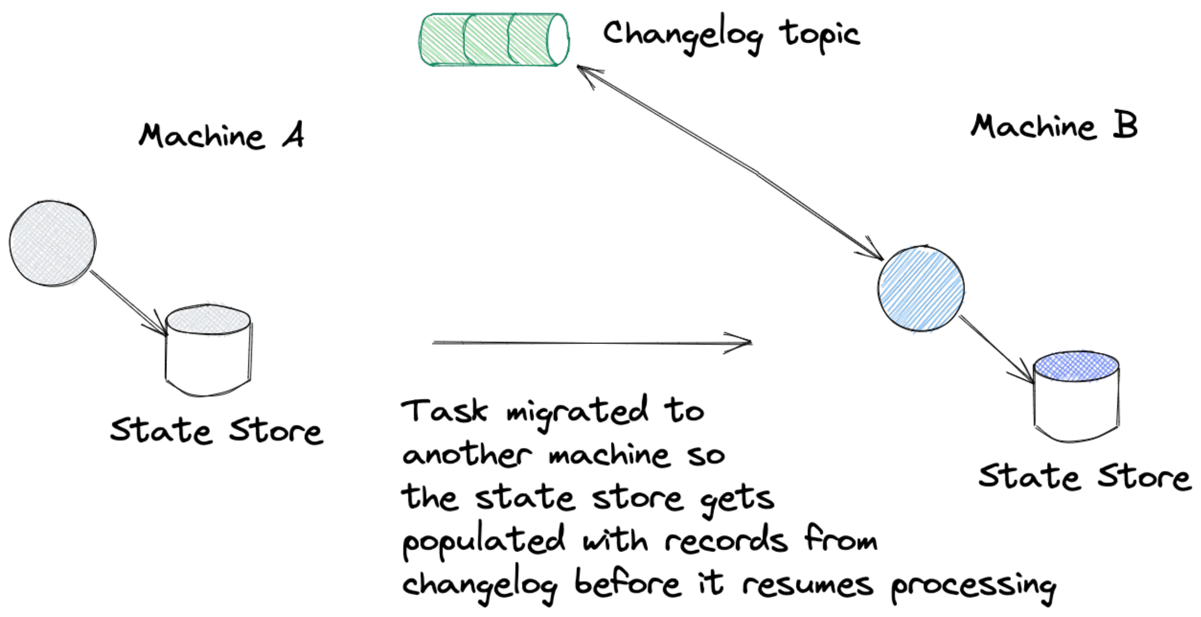
Changelog topic
Machine A
Machine B
State Store
State Store
Task migrated to another machine so the state store gets populated with records from changelog before it resumes processing

Figure 7.18 When a stateful task gets moved to a new machine Kafka Streams rebuilds the state store from the beginning of the changelog topic

So by using changelog topics you can be assured your applications will have a high degree of data durability even in the face of application loss, but there's delayed processing until the store is fully online. Fortunately, Kafka Streams offers a remedy for this situation, the standby task.

7.4.2 Standby Tasks

To enable fast failover from an application instance dropping out of the group Kafka Streams provides the standby task. A standby task "shadows" an active task by consuming from the changelog topic into a state store local to the standby. Then should the active task drop out of the group, the standby becomes the new active task. But since it's been consuming from the changelog topic, the new active task will come online with minimum latency.

IMPORTANT **To enable standby tasks you need to set the `num.standby.replicas` configuration to a value greater than 0 and you need to deploy N+1 number of Kafka Streams instances (with N being equal to the number of desired replicas). Ideally you'll deploy those Kafka Streams instances on separate machines as well.**

While the concept is straight forward, let's review the standby process by walking through the following illustration:

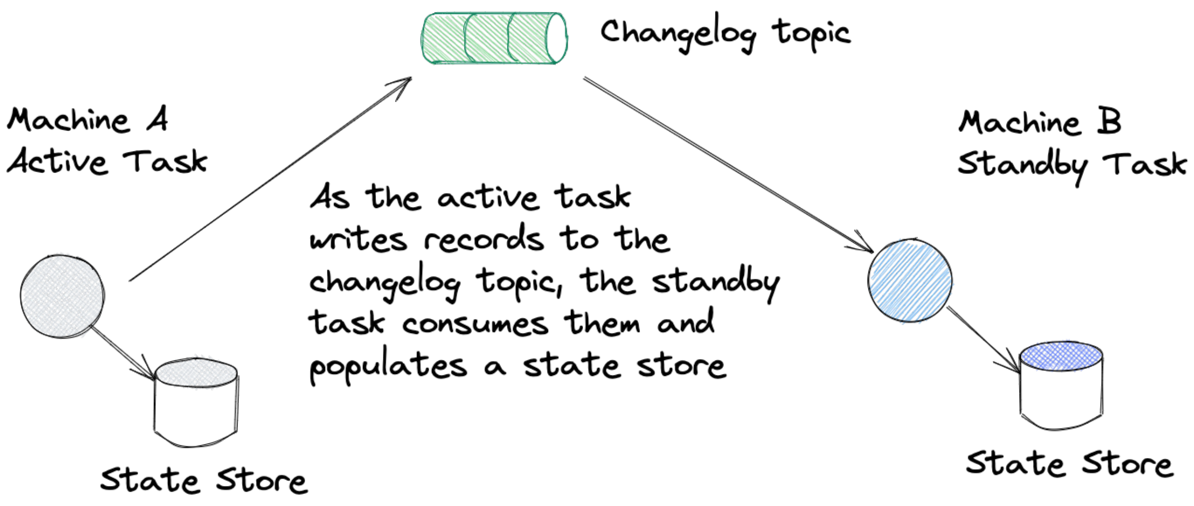
Changelog topic
Machine A
Active Task
Machine B
Standby Task
As the active task writes records to the changelog topic, the standby task consumes them and populates a state store
State Store
State Store

Figure 7.19 A standby task shadows the active task and consumes from the changelog topic keeping a local state store in-sync with store of the active task

So following along with the illustration a standby task consumes records from the changelog topic and puts them in its own local state store. To be clear, a standby task does not process any records, its only job is to keep the state store in sync with the state store of the active task. Just like any standard producer and consumer application, there's no coordination between the active and standby tasks.

With this process since the standby stays fully caught up to the active task or at minimum it will be only a handful of records behind, so when Kafka Streams reassigns the task, the standby becomes the active task and processing resumes with minimal latency as its already caught up. As with anything there is a trade-off to consider with standby tasks. By using standby's you end up duplicating data, but with benefit of near immediate fail-over, depending on your use case it's definitely worth consideration.

NOTE **Significant work went into improving the scaling out performance of Kafka Streams with Kafka KIP-441 (cwiki.apache.org/confluence/display/KAFKA/KIP-441%3A+Smooth+Scaling +Out+for+Kafka+Streams). When you enable standby tasks and the standby instance becomes the active one, if at a later time Kafka Streams determines a more favorable assignment is possible, then that stateful task may get migrated to another instance.**

So far we've covered how state stores enable stateful operations and how the stores are robust due to changelog topic and using standby tasks to enable quick failover. But we still have some more ground to cover. First we'll go over state store assignment, from there you'll learn how to configure state stores including how to specify a store type including an in-memory store and finally how to configure changelog topics if needed.

7.4.3 Assigning state stores in Kafka Streams

In the previous chapter we discussed the role of tasks in Kafka Streams. Here I want to reiterate that tasks operate in a shared nothing architecture and only operate in a single thread. While a Kafka Streams application can have multiple threads and each thread can have multiple tasks, there is nothing shared between them. I emphasize this "shared nothing" architecture again, because this means that when a task is stateful, only the owning task will access its state store, there are no locking or concurrency issues.

Going back to the [Stock-Aggregation] example, let's say the source topic has two partitions, meaning it has two tasks. Let's look at an updated illustration of tasks assignment with state stores for that example:

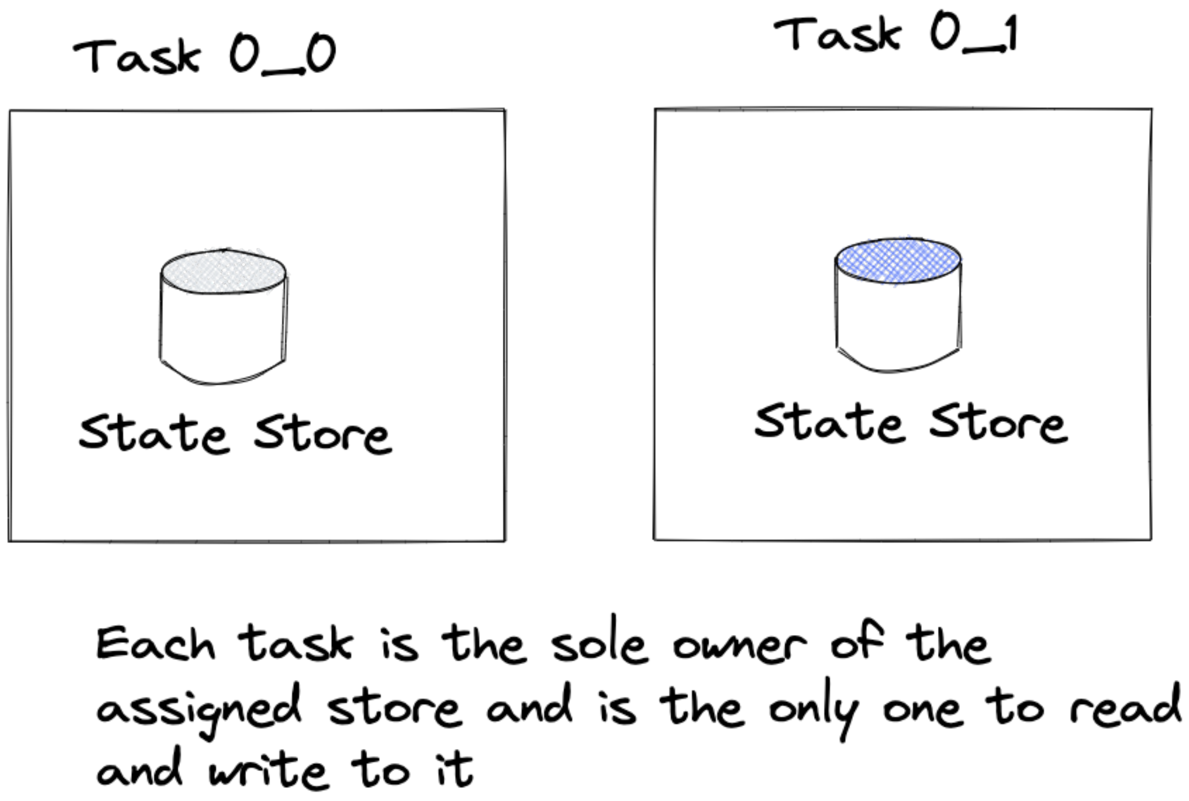
Task 0_0
Task 0_1
State Store
State Store
Each task is the sole owner of the assigned store and is the only one to read and write to it

Figure 7.20 Stateful tasks have a state store assigned to it

By looking at this illustration you can see that the task associated with the state store is the only task that will ever access it. Now let's talk about how Kafka Streams places state stores in the file system.

7.4.4 State store location on the file system

When you have a stateful application, when Kafka Streams first starts up, it creates a root directory for all state stores from the `StreamsConfig.STATE_DIR_CONFIG` configuration. If not set, the `STATE_DIR_CONFIG` defaults to the temporary directory for the JVM followed by the system dependent separator and then "kafka-streams".

IMPORTANT **The value of the `STATE_DIR_CONFIG` configuration must be unique for each Kafka Streams instance that shares the same file system**

For example on my MacOS the default root directory for state stores is `/var/folders/lk/d_9__qr558zd6ghbqwty0zc80000gn/T/kafka-streams`.

TIP **To view the system dependent temporary directory on you machine you can start a Java shell from a terminal window by running the `jshell` command. Then type in `System.getProperty("java.io.tmpdir")`, hit the `return` key and it will display on the screen.**

Next Kafka Streams appends the application-id, which you have to provide in the configurations, to the path. Again on my laptop the path is `/var/folders/lk/d_9__qr558zd6ghbqwty0zc80000gn/T/kafka-streams/test-application/`

From here the directory structure branches out to unique directories for each task. Kafka Streams creates a directory for each stateful task using the subtopology-id and partition (separated by an underscore) for the directory name. For example a stateful task from the first subtopology and assigned to partition zero would use `0_0` for the directory name.

The next directory is named for the implementation of the store which is `rocksdb`. So at this point the path would look like

```
/var/folders/lk/d_9__qr558zd6ghbqwty0zc80000gn/T/kafka-streams/test-
application/0_0/rocksdb
```

. It is under this directory there is the final directory from the processor (unless provided by a `Materialized` object and I'll cover that soon). To understand how the final directory gets its name, let's look at snippet of a stateful Kafka Streams application and the generated topology names.

Listing 7.17 Simple Kafka Streams stateful application

```
builder.stream("input")
       .groupByKey()
       .count()
       .toStream()
       .to("output")
```

This application has topology named accordingly: .Topology names

```
Topologies:
   Sub-topology: 0
    Source: KSTREAM-SOURCE-0000000000 (topics: [input])
      --> KSTREAM-AGGREGATE-0000000002
    Processor: KSTREAM-AGGREGATE-0000000002  ❶
     (stores: [KSTREAM-AGGREGATE-STATE-STORE-0000000001])  ❷
      --> KTABLE-TOSTREAM-0000000003
      <-- KSTREAM-SOURCE-0000000000
    Processor: KTABLE-TOSTREAM-0000000003 (stores: [])
      --> KSTREAM-SINK-0000000004
      <-- KSTREAM-AGGREGATE-0000000002
    Sink: KSTREAM-SINK-0000000004 (topic: output)
      <-- KTABLE-TOSTREAM-0000000003
```

❶ The name of the aggregate processor

❷ The name of the store assigned to processor

From the topology here Kafka Streams generates the name `KSTREAM-AGGREGATE-0000000002` for the `count()` method and notice it's associated with the store named `KSTREAM-AGGREGATE-STATE-STORE-0000000001`. So Kafka Streams takes the base name of the stateful processor and appends a `STATE-STORE` and the number generated from the global counter. Now lets take a look at the full path you would find this state store: `/var/folders/lk/d_9__qr558zd6ghbqwty0zc80000gn/T/kafka-streams/test-application/0`

So it's the final directory `KSTREAM-AGGREGATE-STATE-STORE-0000000001` in the path that contains the RocksDB files for that store. Now if you were to check the topics on the broker after starting the Kafka Streams application you'd see this name in the list `test-application-KSTREAM-AGGREGATE-STATE-STORE-0000000001-changelog`. This topic is the changelog for the state store and notice how Kafka Streams uses a naming convention of <application-id>-<state store name>-changelog for the topic.

7.4.5 Naming Stateful operations

This naming raises an interesting question, what happens if we add an operation before the `count()`? Let's say you want to add a filter to exclude certain records from the counting. You'd simply update the topology like so:

Listing 7.18 Updated Topology with a filter

```
builder.stream("input")
       .filter((key, value) -> !key.equals("bad"))
       .groupByKey()
       .count()
       .toStream()
       .to("output")
```

Remember, Kafka Streams uses a global counter for naming the processor nodes, so since you've added an operation, every processor downstream of it will have a new name since the number will be greater by 1. Here's what the new topology will look like:

Listing 7.19 Updated Topology names

```
Topologies:
   Sub-topology: 0
    Source: KSTREAM-SOURCE-0000000000 (topics: [input])
      --> KSTREAM-FILTER-0000000001
    Processor: KSTREAM-FILTER-0000000001 (stores: [])
      --> KSTREAM-AGGREGATE-0000000003
      <-- KSTREAM-SOURCE-0000000000
    Processor: KSTREAM-AGGREGATE-0000000003     ❶
     (stores: [KSTREAM-AGGREGATE-STATE-STORE-0000000002])    ❷
      --> KTABLE-TOSTREAM-0000000004
      <-- KSTREAM-FILTER-0000000001
    Processor: KTABLE-TOSTREAM-0000000004 (stores: [])
      --> KSTREAM-SINK-0000000005
      <-- KSTREAM-AGGREGATE-0000000003
    Sink: KSTREAM-SINK-0000000005 (topic: output)
      <-- KTABLE-TOSTREAM-0000000004
```

❶ The new name for the aggregation operation

❷ The new name for the state store

Notice how the state store name has changed which means there is a new directory named `KSTREAM-AGGREGATE-STATE-STORE-0000000002` and the corresponding changelog topic is now named `test-application-KSTREAM-AGGREGATE-STATE-STORE-0000000002-changelog`.

NOTE **Any changes before a stateful operation could result in the generated name shift, i.e. removing operators will have the same shifting effect.**

What does this mean to you? When you redeploy this Kafka Streams application the directory will only contain some basic RocksDB file, but not your original contents they are in the previous state store directory. Normally an empty state store directory does not present an issue, as Kafka Streams will restore it from the changelog topic. Except in this case the changelog topic is also new, so it's empty as well. So while your data is still safe in Kafka, the Kafka Streams application will start over with empty state store due to the name changes.

While it's possible to reset the offsets and process data again, a better approach is to avoid name shifting situation all together by providing a name for the state store instead of relying on the generated one. In the previous chapter I covered naming processor nodes for providing a better understanding of what the topology does. But in this case it goes beyond better understanding of its role in the topology, which is important, but also makes your application robust in the face of a changing topology.

Going back to the simple `count()` example in this section, you'll update the application by passing `Materialized` object to `count()` operation:

Listing 7.20 Naming the state store using a Materialized object

```
builder.stream("input")
       .groupByKey()
       .count(Materialized.as("counting-store"))   ❶
       .toStream()
       .to("output")
```

❶ Explicitly naming the state store

By providing the name of the state store, Kafka Streams will name the directory on disk `counting-store` and the changelog topic becomes `test-application-counting-store-changelog`, and both of these names are "frozen" and will not change regardless of any updates you make to the topology. It's important to note that the names of state stores within a topology must be unique, otherwise you'll get a `TopologyException`.

NOTE **Only stateful operations are affected by name shifting. But since stateless operations don't keep any state, changes in processor names from topology updates will have no impact.**

The bottom line is to ***always*** name state stores and repartition topics using the appropriate configuration object. By naming the stateful parts of your applications, you can ensure that topology updates don't break the compatibility. Here's a table summarizing which configuration object to use and the operation(s) it applies to:

Table 7.1 Kafka Streams configuration objects for naming state stores and repartition topics

Configuration Object	What's Named	Where Used
Materialized	State Store, Changelog topic	Aggregations
Repartitioned	Repartition topic	Repartition (manual by user)
Grouped	Repartition topic	GroupBy (automatic repartitioning)
StreamJoined	State Store, Changelog topic, Repartition topic	Joins (automatic repartitioning)

Naming state stores provides the added benefit of being able to query them while your Kafka Streams application is running, providing live, materialized views of the streams. I'll cover interactive queries in the next chapter.

So far you've learned how Kafka Streams uses state stores in support of stateful operations. You also learned that the default is for Kafka Streams to use persistent stores and there are in-memory store implementations available. In the next section I'm going to cover how you can specify a different store type as well as configuration options for the changelog topics.

7.4.6 Specifying a store type

All the examples so far in this chapter use persistent state stores, but I've stated that you can use in-memory stores as well. So the question is how do you go about using an in-memory store? So far you've used the `Materialized` configuration object to specify `Serdes` and the name for a store, but you can use it to provide a custom `StateStore` instance to use. Kafka Streams makes it easy to provide an in-memory version of the available store types (so far I've only covered "vanilla" key-value stores, but I'll get to sessioned, windowed, timestamped stores in the next chapter).

The best way to learn how to use a different store type is to change one of our existing examples. Let's revisit the first stateful example used to keep track of scores in an online poker game:

Listing 7.21 Performing a reduce in Kafka Streams to show running total of scores in an online poker game updated to use in-memory stores

```
KStream<String, Double> pokerScoreStream = builder.stream("poker-game",
        Consumed.with(Serdes.String(), Serdes.Double()));

pokerScoreStream
        .groupByKey()
        .reduce(Double::sum,
                Materialized.<String, Double>as(
            Stores.inMemoryKeyValueStore("memory-poker-score-store"))   ❶
                  .withKeySerde(Serdes.String())   ❷
                  .withValueSerde(Serdes.Double()))   ❸
        .toStream()
        .to("total-scores",
                Produced.with(Serdes.String(), Serdes.Double()));
```

❶ Passing a `StoreSupplier` to specify an in-memory store

❷ Specifying the `Serdes` for the key

❸ Specifying the `Serdes` for the value

So by using the overloaded `Materialized.as` method, you provide a `StoreSupplier` using one of the factory methods available from the `Stores` class. Notice that you still pass the serde instances needed for the store. And that's all it takes to switch the store type from persistent to in-memory.

NOTE **Switching in a different store type is fairly straight forward so I'll only have the one example here. But the source code will contain a few additional examples.**

So why would you want to use an in-memory store? Well, an in-memory store will give you faster access since it doesn't need to go to disk to retrieve values. So a topology using in-memory stores should have higher throughput than one using persistent ones. But there are trade-offs you should consider.

First, an in-memory store has limited storage space, and once it reaches it's memory limit it will evict entries to make space. The second consideration is when you stop and restart a Kafka Streams application, under "happy-path" conditions, the one with persistent stores will start processing faster due to the fact that it will have all its state already, but the in-memory stores will always need to restore from the changelog topic.

Kafka Streams provides a factory class `Stores` that provides methods for creating either `StoreSuppliers` or `StoreBuilders`. The choice of which one to use depends on the Kafka Streams API. When using the DSL you'll use `StoreSuppliers` with a `Materialized` object. In the Processor API, you'll use a `StoreBuilder` and directly add it to the topology. I'll cover the Processor API in chapter 9.

TIP **To see all the different store types you can create view the JavaDoc for the `Stores` class javadoc.io/doc/org.apache.kafka/kafka-streams/latest/org/apache/kafka/streams/state/Stores.html**

Now that you've learned how to specify a different store type, let's move on to one more topic to cover with state stores, how you can configure the changelog topic.

7.4.7 Configuring changelog topics

There's nothing special about changelog topics, so you can use any configuration parameters available for topics. But for the most part the default settings should suffice, so you should only consider changing the configurations when it's absolutely necessary.

NOTE **State store changelogs are compacted topics, which we discussed in chapter 2. As you may recall, the delete semantics require a null value for a key, so if you want to remove a record from a state store permanently, you'll need to do a put(key, null) operation.**

Let's revisit the example from above where you provided a custom name for the state store. Let's say the data processed by this application also has a large key space. The changelogs in Kafka Streams are ***compacted*** topics. Compacted topics use a different approach to cleaning up older records.

Instead of deleting log segments by size or time, log segments are *compacted* by keeping only the latest record for each key—older records with the same key are deleted. But since the key space is large compaction may not be enough, as the size of the log segment will keep growing. In that case, the solution is simple. You can specify a cleanup policy of `delete` and `compact` .

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/kafka-streams-in-action-second-edition/discussion

Listing 7.22 Setting a cleanup policy and using Materialized to set the new configuration

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

builder.stream("input")
       .groupByKey()
       .count(Materialized.as("counting-store")
              .withLoggingEnabled(changeLogConfigs))   ❶
       .toStream()
       .to("output")
```

❶ Using the withLoggingEnabled method to set a configuration

So here you can adjust the configurations for this specific changelog topic. Earlier I mentioned that to disable the caching that Kafka Streams uses for stateful operations, you'd set the `StreamsConfig.CACHE_MAX_BYTES_BUFFERING_CONFIG` setting to zero. But since it's in the configuration, it is globally applied to all stateful operations. If you only wanted to disable the cache for a specific one you could disable it by calling `Materialized.withCachingDisabled()` method when passing in the `Materialzied` object.

WARNING **The `Materialized` object also provides a method to disable logging. Doing so will cause the state store to not have a changelog topic, hence it is subject to getting in a state where it can't restore its previous contents. It is recommended to only use this method if absolutely necessary. In my time working with Kafka Streams, I can't say I've encountered a good reason for using this method.**

7.5 Summary

- Stream processing needs state. Stateless processing is acceptable in a lot of cases, but to make more complex decisions you'll need to use stateful operations.
- Kafka Streams provides stateful operations reduce, aggregation, and joins. The state store is created automatically for you and by default they use persistent stores.
- You can choose to use in-memory stores for any stateful operation by passing a `StoreSupplier` from the `Stores` factory class to the `Materialized` configuration object.
- To perform stateful operations your records need to have valid keys-if your records don't have a key or you'd like to group or join records by a different key you can change it and Kafka Streams will automatically repartition the data for you.
- It's important to always provide a name for state stores and repartition topics-this keeps your application resilient from breaking when you make topology changes.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/kafka-streams-in-action-second-edition/discussion