8 Advanced stateful concepts

This chapter covers

- Changelog streams, the KTable and the GlobalKTable
- Aggregating records with a KTable
- Joining a KTable with KStream or another KTable
- Windowing to capture aggregations in specific period of time
- Using suppression for final windowed results
- Understanding the importance of timestamps in Kafka Streams

In this chapter, we're going to continue working with state in a Kafka Streams application. You'll learn about the `KTable` which is considered an update or changelog stream. As a matter of fact you've already used a `KTable` as any aggregation operations in Kafka Streams result in a `KTable`. The `KTable` is an important abstraction for working with records that have the same key. Unlike the `KStream` where records with the same key are still considered independent event, in the `KTable` a record is an update to the previous record that has the same key.

To make a comparison to a relational database, the event stream (a `KStream`) could be considered a series of inserts where the primary key is an auto-incriminating number. As a result each insert of a new record has no relationship to previous ones. But with a `KTable` the key in the key-value pair is the primary key, so each time a record arrives with the same key, it's consider an update to the previous one.

From there you'll learn about aggregation operations with a `KTable`. Aggregations work a little differently because you don't want to group by primary key, you'll only ever have on record that way, instead you'll need to consider how you want to group the records to calculate the aggregate.

Since you can use the `KTable` as lookup table, a join between a stream and table is a power combination, where you can enrich the event stream records by performing a lookup in the table for additional details. You can also join two tables together, even using a foreign key. You'll also learn about a unique construct called the `GlobalKTable` which, unlike the `KTable` which sharded by partitions, contains all records from it's underlying source across all application instances.

After covering the table abstractions we'll get into how to "bucket" your aggregations into specific time periods using windowing. For example, how many purchases have there been over the past hour, updated every ten minutes? Windowing allows you to place data in discrete blocks of time, as opposed to having an unbounded collection. You'll also learn how to produce a single final result from a windowed operation when the window closes. There's also the ability to expose the underlying state stores to queries from outside the application allowing for real-time updates on the information in the event stream.

Our final topic for the chapter is how timestamps drive the behavior in Kafka Streams and especially their impact on windowing and stateful operations.

8.1 KTable The Update Stream

To fully understand the concept of an update stream, it will be useful to compare with an event stream to see the differences between the two. Let's use a concrete example of tracking stock price updates.

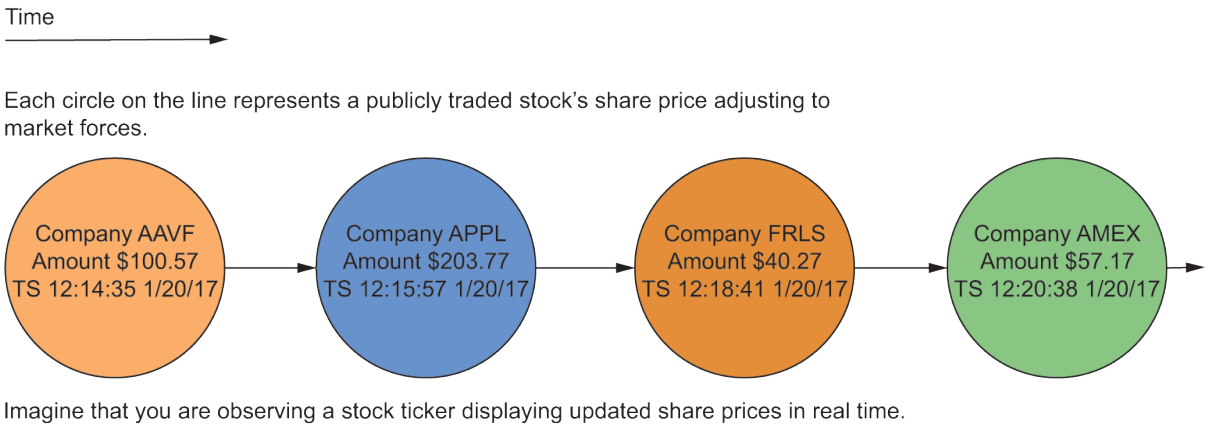
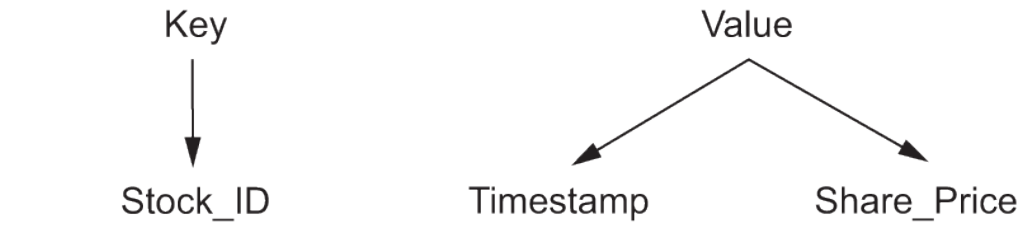


Figure 8.1 A diagram for an unbounded stream of stock quotes

You can see that each stock price quote is a discrete event, and they aren't related to each other. Even if the same company accounts for many price quotes, you're only looking at them one at a time. This view of events is how the `KStream` works—it's a stream of records.

Now, let's see how this concept ties into database tables. Each record is an insert into the table, but the primary key is a number increment for each insert, depicted simple stock quote table in figure 8.2.



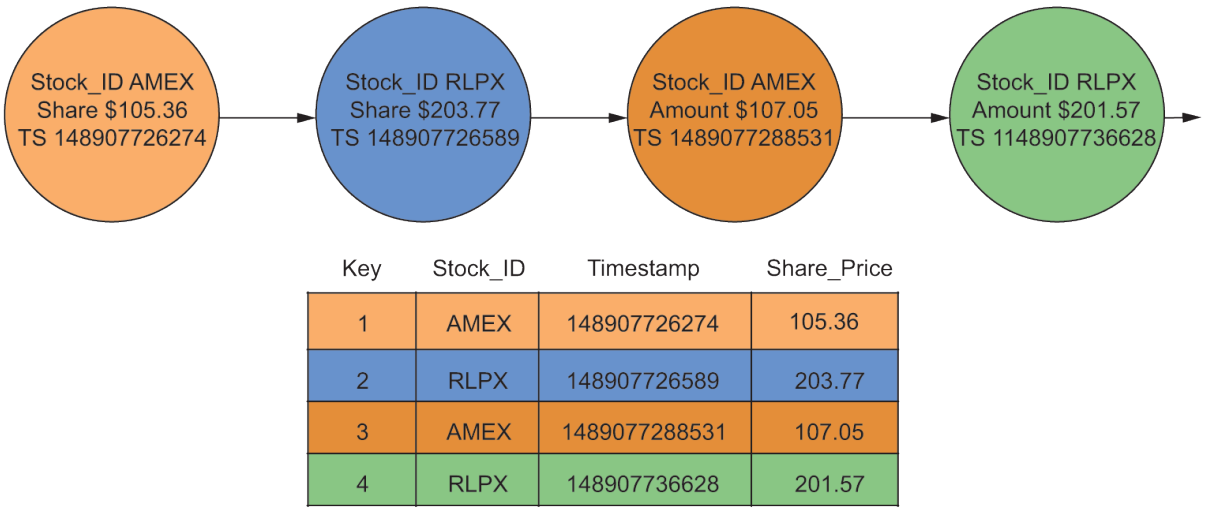
ABVF	32225544289	105.36
APPL	32225544254	333.66

The rows from table above can be recast as key/value pairs. For example, the first row in the table can be converted to this key/value pair:

{key:{stockid:1235588}, value:{ts:3225544289, price:105.36}}

Figure 8.2 A simple database table represents stock prices for companies. There's a key column, and the other columns contain values. You can consider this a key/value pair if you lump the other columns into a "value" container.

Next, let's take another look at the record stream. Because each record stands on its own, the stream represents inserts into a table. Figure 5.3 combines the two concepts to illustrate this point.



Key	Stock_ID	Timestamp	Share_Price
1	AMEX	148907726274	105.36
2	RLPX	148907726589	203.77
3	AMEX	1489077288531	107.05
4	RLPX	148907736628	201.57

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 you consider each item on the stream as a singular event.

As a result, each event is an insert, and you 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.

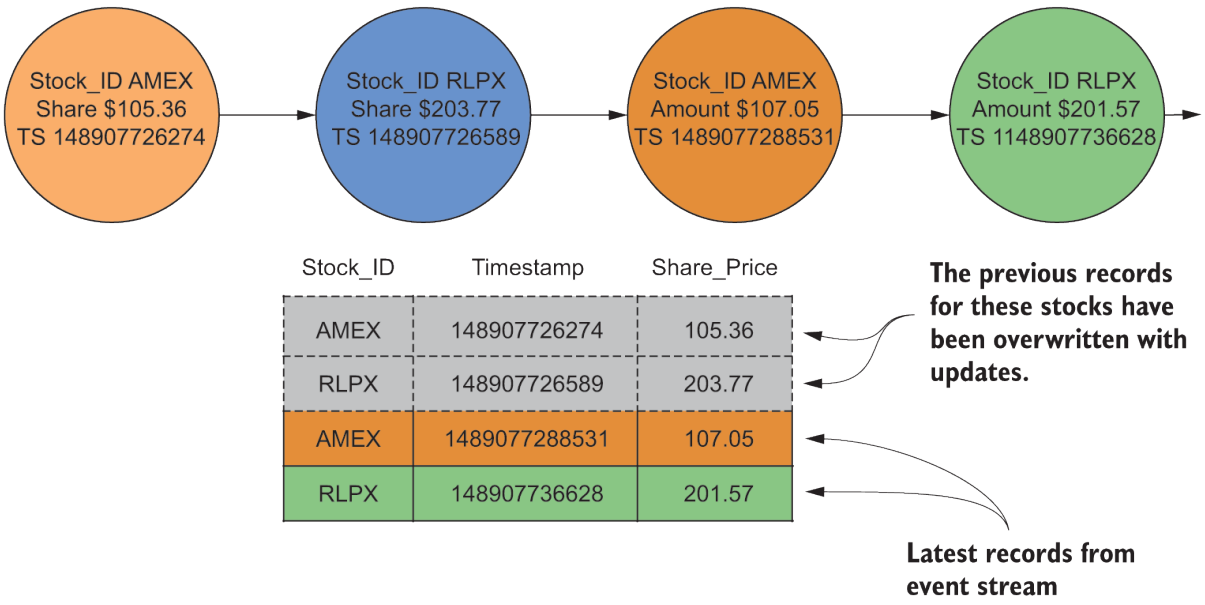
Figure 8.3 A stream of individual events compares to inserts into a database table. You could similarly imagine streaming each row from the table.

What's important here is that you can view a stream of events in the same light as inserts into a table, which can help give you a deeper understanding of using streams for working with events. The next step is to consider the case where events in the stream *are* related to one another.

8.1.1 Updates to records or the changelog

Let's say you want to track customer purchase behavior, so you take the same stream of customer transactions, but now track activity over time. If you add a key of customer ID, the purchase events can be related to each other, and you'll have an update stream as opposed to an event stream.

If you consider the stream of events as a log, you can consider this stream of updates as a changelog. Figure 8.4 demonstrates this concept.



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 8.4 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.

Here, you can see the relationship between a stream of updates and a database table. Both a log and a changelog represent incoming records appended to the end of a file. In a log, you see all the records; but in a changelog, you only keep the latest record for any given key.

> **NOTE** **With both a log and a changelog, records are appended to the end of the file as they come in. The distinction between the two is that in a log, you want to see all records, but in a changelog, you only want the latest record for each key.**

To trim a log while maintaining the latest records per key, you can use log compaction, which we discussed in chapter 2. You can see the impact of compacting a log in figure 8.5. Because you only care about the latest values, you can remove older key/value pairs.[6]

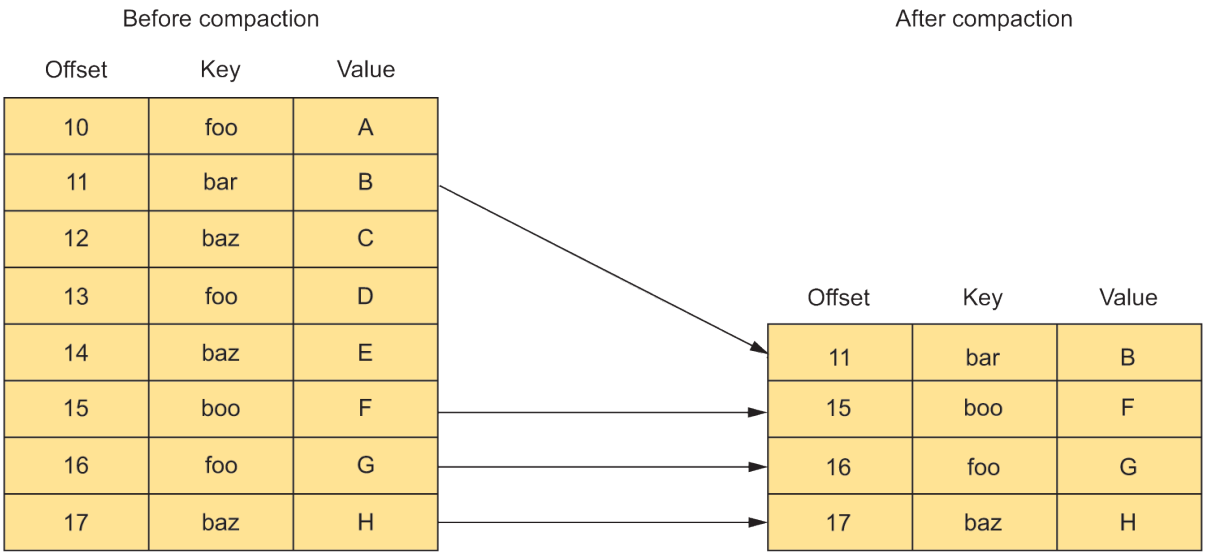


Figure 8.5 On the left is a log before compaction—you'll notice duplicate keys with different values, which are updates. On the right is the log after compaction—you keep the latest value for each key, but the log is smaller in size.

You're already familiar with event streams from working with `KStreams`. For a changelog or stream of updates, we'll use an abstraction known as the `KTable`. Now that we've established the relationship between streams and tables, the next step is to compare an event stream to an update stream.

8.1.2 Event streams vs. update streams

We'll use the `KStream` and the `KTable` to drive our comparison of event streams versus update streams. We'll do this by running a simple stock ticker application that writes the current share price for three (fictitious!) companies. It will produce three iterations of stock quotes for a total of nine records. A `KStream` and a `KTable` will read the records and write them to the console via the `print()` method.

> **NOTE** **The `KTable` does not have methods like `print()` or `peek()` in its API, so to do any printing of records you'll need to convert the `KTable` from an update stream to an event stream by using the `toStream()` method first.**

Figure 8.6 shows the results of running the application. As you can see, the `KStream` printed all nine records. We'd expect the `KStream` to behave this way because it views each record individually. In contrast, the `KTable` printed only three records, because the `KTable` views records as updates to previous ones.

A simple stock ticker for three fictitious companies with a data generator producing three updates for the stocks. The KStream printed all records as they were received. The KTable only printed the last batch of records because they were the latest updates for the given stock symbol.

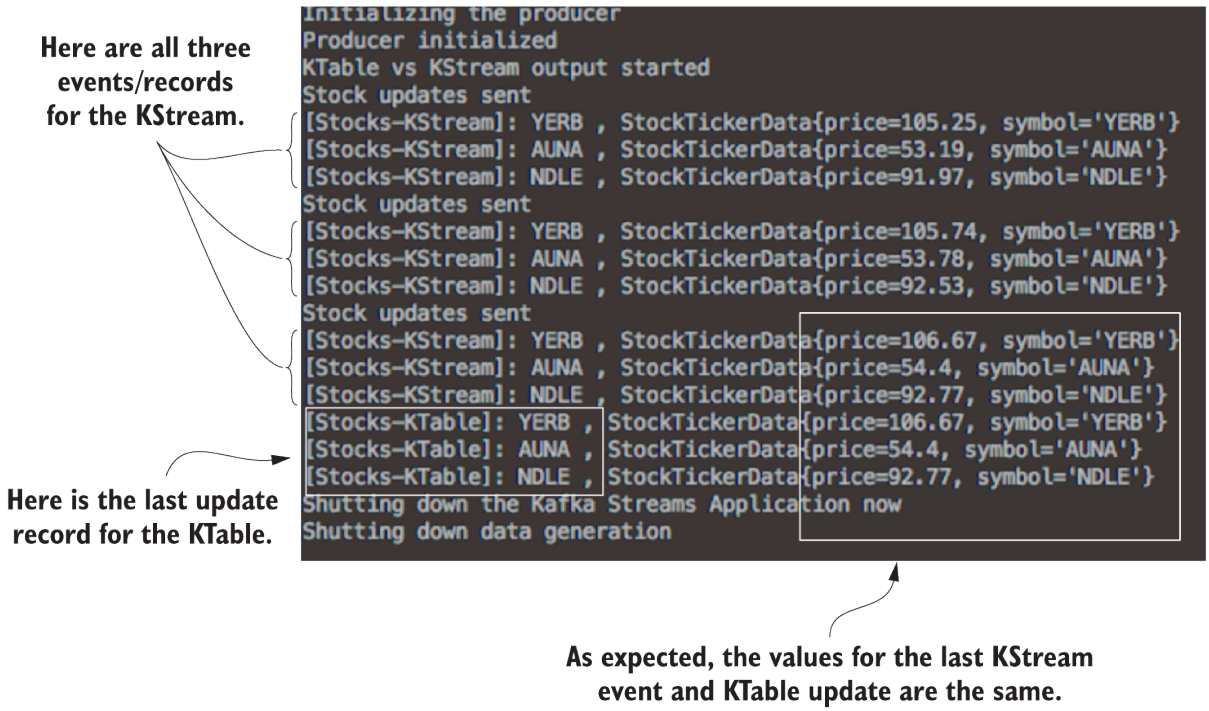


Figure 8.6 `KTable` versus `KStream` printing messages with the same keys

From the `KTable`'s point of view, it didn't receive nine individual records. The `KTable` received three original records and two rounds of updates, and it only printed the last round of updates. Notice that the `KTable` records are the same as the last three records published by the `KStream`. We'll discuss the mechanisms of how the `KTable` emits only the updates in the next section.

Here's the program for printing stock ticker results to the console (found in src/main/java/bbejeck/chapter_8/KStreamVsKTableExample.java; source code can be found on the book's website here: manning.com/books/kafka-streams-in-action-second-edition).

Listing 8.1 `KTable` and `KStream` printing to the console

```
KTable<String, StockTickerData> stockTickerTable =
builder.table(STOCK_TICKER_TABLE_TOPIC); ❶
KStream<String, StockTickerData> stockTickerStream =
builder.stream(STOCK_TICKER_STREAM_TOPIC); ❷

stockTickerTable.toStream()
  .print(Printed.<String, StockTickerData>toSysOut()
  .withLabel("Stocks-KTable")); ❸

stockTickerStream
  .print(Printed.<String, StockTickerData>toSysOut()
  .withLabel("Stocks-KStream")); ❹
```

❶ Creates the KTable instance

❷ Creates the KStream instance

❸ KTable prints results to the console

❹ KStream prints results to the console

SIDEBAR **Using default serdes**

In creating the `KTable` and `KStream`, you didn't specify any serdes to use. The same is true with both calls to the `print()` method. You were able to do this because you registered a default serdes in the configuration. like so:

```
props.put(StreamsConfig.DEFAULT_KEY_SERDE_CLASS_CONFIG,
   Serdes.String().getClass().getName());
props.put(StreamsConfig.DEFAULT_VALUE_SERDE_CLASS_CONFIG,
   StreamsSerdes.StockTickerSerde().getClass().getName());
```

If you used different types, you'd need to provide serdes in the overloaded methods for reading or writing records.

The takeaway here is that records in a stream with the same keys are updates, not new records in themselves. A stream of updates is the main concept behind the `KTable`, which is the backbone of stateful operations in Kafka Streams.

8.2 KTables are stateful

In the previous example when you created the table with the `StreamsBuilder.table` statement Kafka Streams also creates a `StateStore` for tracking the state and by default it's a persistent store. Since state stores only work with byte arrays for the keys and values you'll need to provide the `Serde` instances so the store can (de)serialize the keys and values. Just as you can provide specific serdes to an event stream with `Consumed` configuration object, you can do the same when creating a `KTable`:

```
builder.table(STOCK_TICKER_TABLE_TOPIC,
              Consumed.with(Serdes.String(),
                            StockTradeSerde()));
```

Now the serdes you've provided with the `Consumed` object get passed along to the state store. There's an additional overloaded version of `StreamsBuilder.table` that accepts a `Materialized` instance as well. This allows you to customize the type of store and provide a name to make it available for querying. We'll discuss interactive queries later in this chapter.

It's also possible to create a `KTable` directly by using the `KStream.toTable` method. Using this method changes the interpretation of the records from events to updates. You can also use the `KTable.toStream` method to convert the update stream into an event stream. We'll talk more about this conversion from update stream to event stream when we discuss the `KTable` API.

The main point here is you are creating a `KTable` directly from a topic, which results in creating a state store.

So far I've talked about how the `KTable` handles inserts and updates, but what about when you need to delete a record? To remove a record from a `KTable` you send send a key-value pair with the value set to `null` and this will act as a tombstone marker ultimately getting removed from the state store and the changelog topic backing the store, in other words it's deleted from the table.

Just like the `KStream`, the `KTable` is spread out over tasks determined by the number of partitions in the underlying source topic, meaning that the records for the table are potentially distributed over separate application instances. We'll see a little later in this chapter an abstraction where all the records are available in a single table.

8.3 The KTable API

The `KTable` API offers similar methods to what you'd see with the `KStream - filter`, `filterNot`, `mapValues`, and `transformValues` (I won't talk about `transformValues` here, but we'll cover it in Processor API chapter later in the book). Executing these methods also follow the fluent pattern, they return a new `KTable` instance.

While the functionality of these methods are very similar as the same methods in the `KStream` API, there are some differences in how they operate. The differences come into play due to the fact that key-value pairs where the value is `null` has delete semantics.

So the delete semantics have the following effects on how the `KTable` operates:

1. If the incoming value is null the processor is not evaluated at all and the key-value with the `null` is forwarded to the new table as a tombstone marker.
2. In the case of the `filter` and `filterNot` methods records that get dropped a tombstone record is forwarded to the new table as a tombstone marker as well.

As an example to follow along with see the `KTableFilterExample` in the `bbejeck.chapter_8` package. It runs a simple `KTable.filter` example where some of the incoming values are `null` as well as filtering out some of the non-null values. But since we've discussed filtering previously, I won't review the example here and I'll leave up to you as an exercise to do on your own.

Since I've already covered stateless operations in a previous chapter and we've discussed the different semantics of the `KTable`, we'll move on at this point to discuss aggregations and joins.

8.4 KTable Aggregations

Aggregations in the `KTable` operate a little differently than the ones we've seen in the `KStream`, so let's dive in with an example to illustrate. Imagine you build an application to track stocks. You're only interested in the latest price for any given symbol, so using a `KTable` makes sense as that is its default behavior. Additionally, you'd like tp keep track of how different market segments are performing. For example, you'd group the stocks of Google, Apple, and Confluent into the tech market segment. So you'll need to perform an aggregation and group different stocks together by the market segment they belong to. Here's what your `KTable` aggregation would look like:

Listing 8.2 Aggregates with A KTable

```
KTable<String, StockAlertProto.StockAlert> stockTable =
                builder.table("stock-alert",
                 Consumed.with(stringSerde, stockAlertSerde));   ❶

stockTable.groupBy((key, value) ->
                    KeyValue.pair(value.getMarketSegment(), value),
                Grouped.with(stringSerde, stockAlertSerde))   ❷
        .aggregate(segmentInitializer,   ❸
                adderAggregator,                  ❹
                subtractorAggregator,             ❺
                Materialized.with(stringSerde, segmentSerde))
        .toStream()

        .to("stock-alert-aggregate",
                Produced.with(stringSerde, segmentSerde));
```

❶ Creating the original `KTable`

❷ Grouping by the market segment also providing Serdes for the repartition via a `Grouped`

❸ Creating the aggregate

❹ Providing the adder `Aggregator`

❺ Providing the subtractor `Aggregator`

Annotation one is where you create the `KTable` and is what you'd expect to see but annotation two you're performing a `groupBy` and updating the key to be the market segment which will force a repartition of the data. Now this makes sense, since the original key is the stock symbol you're not guaranteed that all stocks from a given market segment reside on the same partition.

But this requirement somewhat hides the fact with a `KTable` aggregation you'll ***always*** need to perform a group-by operation. Why is this so? Remember that with a `KTable`, the incoming key is considered a ***primary key***, and just like in a relational database, grouping by the primary-key always results in a single record - hence not much is provided for an aggregation. So you'll need to group records by another field because the combination of the primary-key and the grouped field(s) will yield results suitable for an aggregation. And similar to the `KStream` API, calling the

`KTable.groupBy` method returns an intermediate table - `KGroupedTable` which you'll use to execute the `aggregate` method.

The second difference occurs with annotations four and five. With the `KTable` aggregations, just like with the `KStream` the first parameter you provide is an `Initializer` instance, to provide the default value for the first aggregation. However you then supply ***two aggregators*** one that adds the new value into the aggregation and the other one ***subtracts*** values from the aggregation for the previous entry with the same key. Let's look at an illustration to help make this process clear:

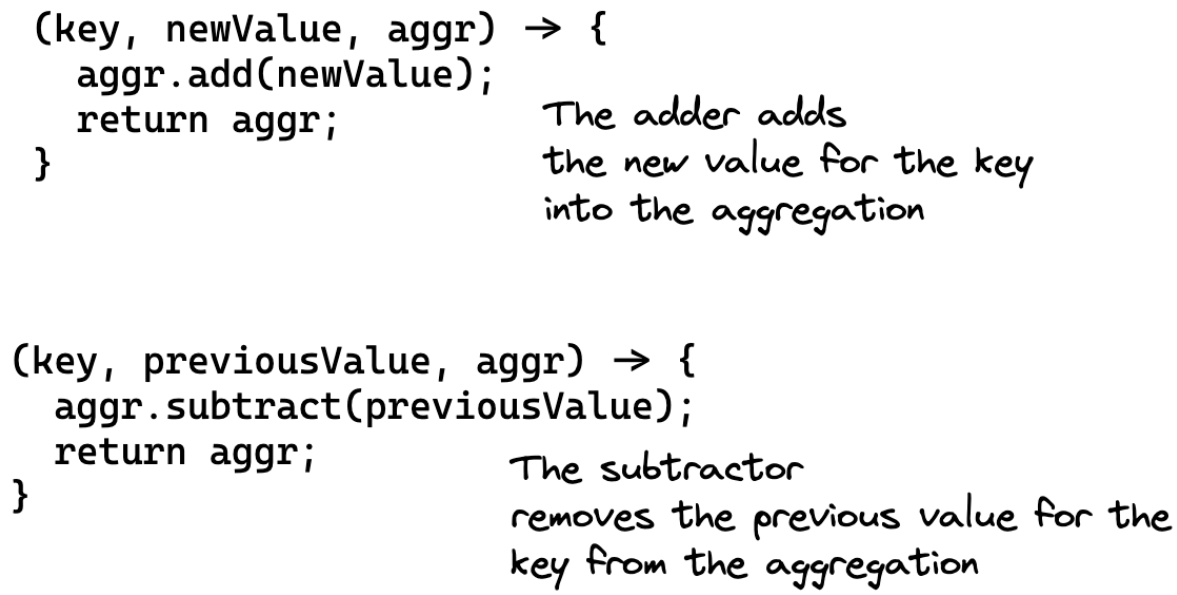


Figure 8.7 KTable Aggregations use an Adder aggregator and a Subtractor aggregator

Here's another way to think about it - if you were to perform the same thing on a relational table, summing the values in the rows created by a grouping, you'd only every get the latest, single value per row created by the grouping. For example the SQL equivalent of this `KTable` aggregation could look something like this:

Listing 8.3 SQL of KTable aggregation

```
SELECT market_segment,
       sum(share_volume) as total_shares,
       sum(share_price * share_volume) as dollar_volume
       FROM stock_alerts
       GROUP BY market_segment;
```

From the SQL perspective, when a new record arrives, the first step is to update the alerts table, then run the aggregation query to get the updated information. This is exactly the process taken by the `KTable`, the new incoming record updates the table for the stock_alerts and it's forwarded to the aggregation. Since you can only have one entry per stock symbol in the roll-up, you add the new record into the aggregation, then remove the previous value for the given symbol.

Consider this example, a record comes in for the ticker symbol CFLT so the `KTable` is updated with new entry. Then the aggregate updates with new entry for CFLT, but since there's already a value for it in the aggregation you must remove it then recalculate the aggregation with the new value.

Now that we've covered how the `KTable` aggregation works, let's take a look at the `Aggregator` instances. But since we've covered them in a previous chapter, let's just take a look at the logic of the adder and subtractor. Even though this is just one example the basic principals will be true for just about any `KTable` aggregation.

Let's start with the adder:

Listing 8.4 KTable adder Aggregator

```
//Some details omitted for clarity

final Aggregator<String,
                 StockAlertProto.StockAlert,
                 SegmentAggregateProto.SegmentAggregate> adderAggregator =
                       (key, newStockAlert, currentAgg) -> {

            long currentShareVolume =
                    newStockAlert.getShareVolume();   ❶
            double currentDollarVolume =
                    newStockAlert.getShareVolume() * newStockAlert.getSharePrice();   ❷

            aggBuilder.setShareVolume(currentAgg.getShareVolume() + currentShareVolume);   ❸
            aggBuilder.setDollarVolume(currentAgg.getDollarVolume() + currentDollarVolume);   ❹
}
```

❶ Extracting the share volume from the current `StockAlert`

❷ Calculating the dollar volume for the current `StockAlert`

❸ Setting the total share volume by adding share volume from the latest `StockAlert` to the current aggregate

❹ Setting the total dollar volume by adding calculated dollar volume to current aggregate

Here the logic is very simple: take the share volume from the latest `StockAlert` and add it to the current aggregate, then do the same with the dollar volume (after calculating it by multiplying the share volume by the share price).

NOTE **Protobuf objects are immutable so when updating values we need to create new instances using a builder that is generated for each unique object.**

Now for the subtractor, you guessed it, you'll simply do the reverse and **subtract** the same values/calculations for the previous record with the same stock ticker symbol in the given market segment. Since the signature is the same I'll only show the calculations:

Listing 8.5 KTable subtractor Aggregator

```
//Some details omitted
long prevShareVolume = prevStockAlert.getShareVolume();
double prevDollarVolume =
      prevStockAlert.getShareVolume() * prevStockAlert.getSharePrice();

aggBuilder.setShareVolume(currentAgg.getShareVolume() - prevShareVolume);   ❶
aggBuilder.setDollarVolume(currentAgg.getDollarVolume() - prevDollarVolume);  ❷
```

❶ Subtracting the share volume from the previous `StockAlert`

❷ Subtracting the dollar volume from the previous `StockAlert`

The logic is straight forward, you're subtracting the values from the `StockAlert` that has been replaced in the aggregate. I've added some logging to the example to demonstrate what is going on and it will be a good idea to look over a portion of that now to nail down what's going on:

Listing 8.6 Logging statements demonstrating the adding and subtracting process of the KTable aggregate

```
Adder                                                                                and
  ❶  : -> key textiles stock alert symbol: "PXLW" share_price: 2.52 share_volume: 4     aggregat
Adder                                                                  ❷              e
     market_segment: "textiles" : <- updated aggregate dollar_volume: 10.08 share_volume: 4
Subtractor: -> key textiles stock alert symbol: "PXLW" share_price: 2.52 share_volume: 4
market_segment: "textiles"
                  and aggregate dollar_volume: 54.57 share_volume: 18  ❸
Subtractor: <- updated aggregate dollar_volume: 44.49 share_volume: 14    ❹

Adder     : -> key textiles stock alert symbol: "PXLW" share_price: 3.39 share_volume: 6
          market_segment: "textiles"                                    ❺
                       and aggregate dollar_volume: 44.49 share_volume: 14
Adder     : <- updated aggregate dollar_volume: 64.83 share_volume: 20   ❻
```

❶ First entry and the aggregate is empty

❷ The aggregate now updates with the values from the PXLW stock statistics

❸ As a result of a new entry for PXLW the aggregation runs the subtractor

❹ Returning the updated aggregation minus previous values for PXLW

❺ The incoming entry for the new PXLW stock alert

❻ Returning the updated aggregation with new values added

By looking at this output excerpt you should be able to clearly see how the `KTable` aggregate works, it keeps only the latest value for each unique combination of the original `KTable` key and the key used to execute the grouping, which is exactly what you'd expect, since you're performing an aggregation over a table with only one entry per primary key.

It's worth noting here that `KTable` API also provides `reduce` and `count` methods which you'll take similar steps. You first perform a `groupBy`, and for the `reduce` provide an adder and

subtractor `Reducer` implementation. I won't cover them here, but there will be examples of both `reduce` and `count` in the source code for the book.

This wraps up our coverage of the `KTable` API but before we move on to more advanced stateful subjects, I'd like to go over another table abstraction offered by Kafka Streams, the `GlobalKTable`.

8.5 GlobalKTable

I alluded to the `GlobalKTable` earlier in the chapter when we discussed that the `KTable` is partitioned, hence its distributed out among Kafka Streams application instances (with the same application id of course). What makes the `GlobalKTable` unique is the fact that it's not partitioned, it fully consumes the underlying source topic. This means there is a full copy of all records in the table for all application instances.

Let's look at an illustration to help make this clear:

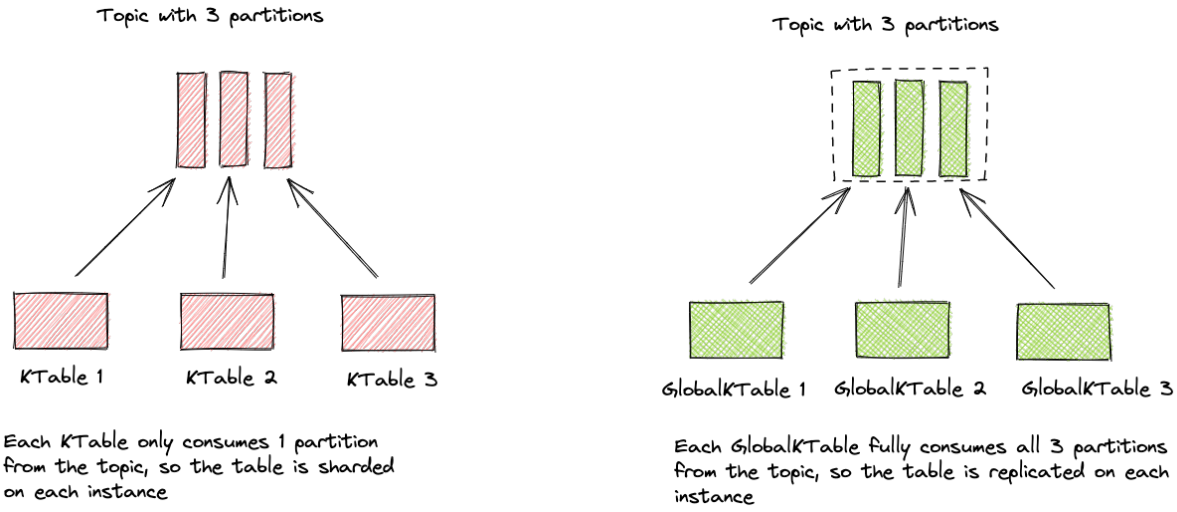


Figure 8.8 GlobalKTable contains all records in a topic on each application instance

As you can see the source topic for the `KTable` has three partitions and with three application instances, each `KTable` is responsible for one partition of data. But the `GlobalKTable` has the **full copy** of its three-partition source topic on each instance. Kafka Streams materializes the `GlobalKTable` on local disk in a `KeyValueStore`, but there is no changelog topic crated for this store as the source topic serves as the backup for recovery as well.

Here's how you'd create one in your application:

Listing 8.7 Creating a GlobalKTable

```
StreamsBuilder builder = new StreamsBuilder();
GlobalKTable<String, String> globalTable =
  builder.globalTable("topic",
                      Consumed.with(Serdes.String(),
                                    Serdes.String()));
```

The interesting thing to note about the `GlobalKTable` is that it doesn't offer an API. So I'm sure you're asking yourself "why would I ever want to use one?". The answer to that question will come in our next section when we discuss joins with the `KTable`.

8.6 KTable Joins

In the previous chapter you learned about performing joins with two `KStream` objects, but you can also perform `KStream-KTable`, `KStream-GlobalKTable`, and `KTable-KTable` joins. Why would you want to join a stream and a table? Stream-table joins represent an excellent opportunity to create an ***enriched*** event with additional information. For the stream-table and table-table joins, both sides need to be co-partitioned - meaning the underlying source topics must have the same number of partitions. If that is not the case then you'll need to do a `repartition` operation to achieve the co-partitioning. Since the `GlobalKTable` has a full copy of the records there isn't a co-partitioning requirement for stream-global table joins.

For example, let's say you have an event stream of user activity on a website, a clickstream, but you also maintain a table of current users logged into the system. The clickstream event object only contains a user-id and the link to the visited page but you'd like more information. Well you can join the clickstream against the user table and you have much more useful information about the usage patterns of your site - in real time. Here's an example to work through:

Listing 8.8 Stream-Table join to enrich the event stream

```
KStream<String, ClickEventProto.ClickEvent> clickEventKStream =
                builder.stream("click-events",
                        Consumed.with(stringSerde, clickEventSerde));

KTable<String, UserProto.User> userTable =
        builder.table("users",
                Consumed.with(stringSerde, userSerde));

clickEventKStream.join(userTable, clickEventJoiner)
        .peek(printKV("stream-table-join"))
        .to("stream-table-join",
                Produced.with(stringSerde, stringSerde));
```

Looking at the code in this example, you first create the click-event stream then a table of logged in users. In this case we'll assume the stream has the user-id for the key and the user tables' primary key is the user-id as well, so we can easily perform a join between them as is. From there you call the `join` method of the stream passing in the table as a parameter.

8.7 Stream-Table join details

At this point I'd like to cover a few of differences with the stream-table joins from the stream-stream join. First of all stream table joins aren't reciprocal - the stream is always on the left or calling side and the table is always on the right side. Secondly, there is no window that the timestamps of the records need to fit into for a join to occur, which dove tails into the third difference; only updates on the stream produce a join result.

In other words, it's only newly arriving records on the stream that trigger a join, new records to the table update the value for the key in table, but don't result in a join result. To capture the join result you provide a `ValueJoiner` object that accepts the value from both sides and produces a new value which can be the same type of either side or a new type altogether. With stream-table joins you can perform an inner (equi) join or a left-outer join (demonstrated here).

8.8 Table-Table join details

Next, let's talk about table-table joins. Joins between the two tables are pretty much the same that you've seen so far with join functionality. Joins between two tables is similar to stream-stream joins, except there is now windowing, but updates to either side will trigger a join result. You provide a `ValueJoiner` instance that calculates the join results and can return an arbitrary type. Also, the constraint that the source topic for both sides have the same number partitions applies here as well.

But there's something extra offered for table-table joins. Let's say you have two `KTables` you'd like to join; users and purchase transactions, but the primary key for the users is user-id and the primary key for transactions is a transaction-id, although the transaction object contains the user-id. Usually a situation like this would require some sort of workaround, but not now, as the `KTable` API offers a foreign-key join, so you can easily join the two tables. To use the foreign-key join you use the signature of the `KTable.join` method that looks like this:

Listing 8.9 KTable Foreign Key join

```
userTable.join(transactionTable,     ❶
               foreignKeyExtractor,  ❷
               joiner);              ❸
```

❶ Other table or right side of the join

❷ The foreign key extractor function

❸ The ValueJoiner parameter

Setting up the foreign key join is done like any other table-table join except that you provide an additional parameter a `java.util.Function` object, that extracts the key used to complete the join. Specifically, the function extracts the key from the left-side value to correspond with the

key of the right side table. If the function returns `null` then no join occurs.

Inner and left-outer joins support joining by a foreign key. As with primary-key table joins, an update on either side will trigger a potential join result. The inner workings of the foreign-key join in Kafka Streams is involved and I won't go into those details, but if you are interested in more details then I suggest reading KIP-213 cwiki.apache.org/confluence/display/KAFKA/KIP-213+Support+non-key+joining+in+KTable.

> **NOTE** **There isn't a corresponding explicit foreign-key joins available in the `KStream` API and that is intentional. The `KStream` API offers methods `map` and `selectKey` where you can easily change the key of a stream to facilitate a join.**

8.9 Stream-GlobaTable join details

The final table join for us to discuss is the stream-global table join. There a few differences with the stream-global table we should cover. First it's the only join in Kafka Streams that does not require co-partitioning. Remember the `GlobalKTable` is not sharded like the `KTable` is, a partition per task, but instead contains all the data of its source topic. So even if the partitions of the stream and the global-table don't match, if the key is present in the global table, a join result will occur.

The semantics of a global table join are different as well. Kafka Streams process incoming `KTable` records along with every other incoming records by timestamps on the records, so with a stream-table join the records are aligned by timestamps. But with a `GlobalKTable`, updates are simply applied when records are available, it's done separately from the other components of the Kafka Streams application.

Having said that, there are some key advantages of using a `GlobalKTable`. In addition to having all records on each instance, stream-global tables support foreign key joins, the key of the stream does not have to match the key of the global table. Let's look at a quick example:

Listing 8.10 KStream GlobalTable Join example

```
userStream.join(detailsGlobalTable,    ❶
                keySelector,    ❷
                valueJoiner);    ❸
```

❶ The `GlobalTable` to join against

❷ A key selector to perform the join

❸ The `ValueJoiner` instance to compute the result

So with the `KStream-GlobalKTable` join the second parameter is a `KeyValueMapper` that takes

the key and value of the stream and creates the key used to join against the global table (in this way it is similar to the `KTable` foreign-key join). It's worth noting that the result of the join will have the key of the stream regardless of the `GlobalTable` key or what the supplied function returns.

Of course every decision involves some sort of trade-off. Using a `GlobalKTable` means using more local disk space and a greater load on the broker since the data is not sharded, but the entire topic is consumed. The stream-global table join is not reciprocal, the `KStream` is always on the calling or left-side of the join. Additionally, only updates on the stream produce a join result, a new record for the `GlobalKTable` only updates the internal state of the table. Finally, either inner or left-outer joins are available.

So what's best to use when joining with a `KStream` a `KTable` or `GlobalKTable`? That's a tough question to answer as there are no hard guidelines to follow. But a good rule of thumb would be to use a `GlobalKTable` for cases where you have fairly static lookup data you want to join with a stream. If the data in your table is large strongly consider using a `KTable` since it will end up being sharded across multiple instances.

At this point, we've covered the different joins available on both the `KTable` and `GlobalKTable`. There's more to cover with tables, specifically viewing the contents of the tables with interactive queries and suppressing output from a table (`KTable` only) to achieve a single final result. We'll cover interactive queries a little later in the chapter. But we'll get to suppression in our next section when we discuss windowing.

8.10 Windowing

So far you've learned about aggregations on both the `KStream` and `KTable`. While they both produce an aggregation, how they are calculated is bit different. Since the `KStream` is an event stream where all records are unrelated, the aggregations will continue to grow over time. But with either case the results produced are cumulative over time. Maybe not as much for `KTable` aggregations, but that's definitely the case for `KStream` aggregations.

There's good chance that you'll want to see results within given time intervals. For example, what's the average reading of a IoT temperature sensor every 15 minutes? To capture aggregations in slices of time, you'll want to use windowing. In Kafka Streams windowing an aggregation means that you'll get results in distinct blocks of time as defined by the size of the window. There are four window types available:

1. Hopping - Windows with a fixed size by time and the advance time is less than the window size resulting in overlapping windows. As a result, results may be included in more than one window. You'd use a hopping window when a result from the previous window is useful for comparison such as fraud detection.
2. Tumbling - A special case of a hopping window where the advance time is the same as

the window size, so each window contains unique results. A good use case for tumbling windows is inventory tracking because you only want the unique amount of items sold per window.

3. Session - A different type of window where its size is not based on time but on behavior instead. Session windows define an inactivity-gap and as long as new events arrive within the defined gap, the window grows in size. But once reaching the inactivity gap, new events will go into a new window. Session windows are great for tracking behavior becuase the windows are determined by activity.
4. Sliding - Sliding windows are fixed time windows, but like the session window, they can continue to grow in size because they are based on behavior as well. Sliding windows specify the maximum difference between timestamps of incoming records for inclusion in the window.

What we'll do next is present some examples and illustrations demonstrating how to add windowing to aggregations and more detail on how they work.

The first example will show how to implement a hopping window on a simple count application. Although the examples will be simple to make learning easier to absorb, the simplicity of the aggregation doesn't matter. You can apply windowing to any aggregation.

Listing 8.11 Setting up hopping windows for an aggregation

```
//Some details omitted for clarity
  countStream.groupByKey()
   .windowedBy(TimeWindows.ofSizeWithNoGrace(Duration.ofMinutes(1))   ❶
            .advanceBy(Duration.ofSeconds(10)))       ❷
   .count(Materialized.as("hopping-window-counting-store"))
   .toStream()            ❸
   .map((windowedKey, value) -> KeyValue.pair(windowedKey.key(), value))   ❹
   .to("counting-output", Produced.with(stringSerde, longSerde));
```

❶ Setting up the window (hopping) with a size of one minute

❷ Establishing the advance by time

❸ Convert the KTable to a KStream

❹ Mapping the Windowed key back to the original inner key

In this example you're using a hopping window with a size of one minute and an advance of 10 seconds. Let's review the code here to understand what your specifying. The `windowedBy` call at annotation one sets up the windowing and specifies the size of the window. You no doubt noticed the method name `ofSizeWithNoGrace`, so what does the `WithNoGrace` mean (other than dribbling your dinner down the front of your shirt!)? Grace is a concept in Kafka Streams that allows you to define how you want to handle out-of-order records, but I'd like to defer that conversation until we've finished discussing the hopping window.

At annotation two, you use the `advanceBy` call which determines the interval that the aggregation will occur. Since the `advanceBy` is less than the window size, it is a hopping window. At annotation three we convert the `KTable` to a `KStream` as we need to convert from

the update stream to an event stream so we can perform some operations on each result.

At annotation four, you use a `map` processor to create a new `KeyValue` object, specifically creating a new key. This new key is actually the original key for the pair when it entered the aggregation. When you perform a windowed aggregation on a `KStream` the key going into the `KTable` gets "upgraded" to a `Windowed` key. The `Windowed` class contains the original key from the record in the aggregation and a reference to the specific `Window` the record belongs to.

Processing the key-values coming from a windowed aggregation presents you with a choice; keep the key as is, or use the `map` processor to set the key the original one. Most of the time it will make sense for you to revert to the original key, but there could be times where you want to keep the `Windowed` one, there really isn't any hard rules here. In the source code there's an example of using both approaches.

Getting back to how a hopping window operates, here's an illustration depicting the action:

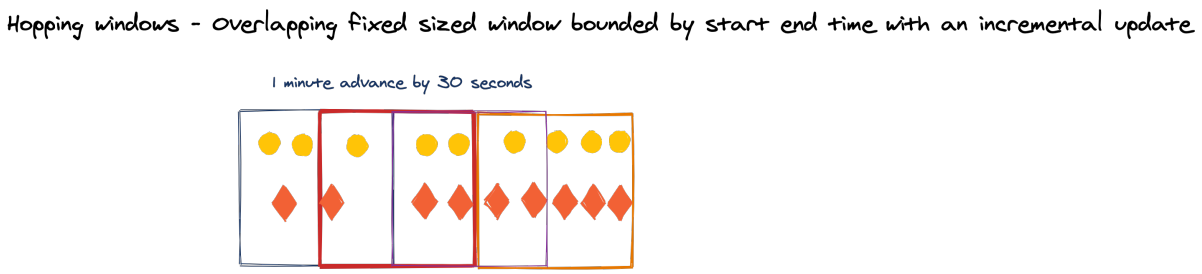


Figure 8.9 Hopping windows hop to the right by the given advance time which

From looking at the illustration, the first record into the aggregation opens the window with the time of its timestamp. Every ten seconds, again based on record timestamps, the aggregation performs its calculation, a simple count in this case. So a hopping window has a fixed size where it collects records, but it doesn't wait the entire time of the window size to perform the aggregation; it does so at intervals within the window corresponding to the advance time. Since the aggregation occurs within the window time it may contain some records from the previous evaluation.

You now have learned about the hopping window, but so far we've assumed that records allways arrive in order. Suppose some of your records don't arrive in order and you'd still like to include them (up to a point) in your count, what whould you do to handle that? Now's a good time to circle back to the concept of out-of-order and grace I mentioned previously.

8.11 Out order records and grace

It will be easier to grasp the concept of grace if we first describe what an out-of-order record is. You've learned in a previous chapter the `KafkaProducer` will set the timestamp on a record. In this case timestamps on the records in Kafka Streams should always increase. But in some cases you may want to use a timestamp embedded in the value, and in that scenario you can't be guaranteed that those timestamps always increase. The following illustration demonstrates this concept:

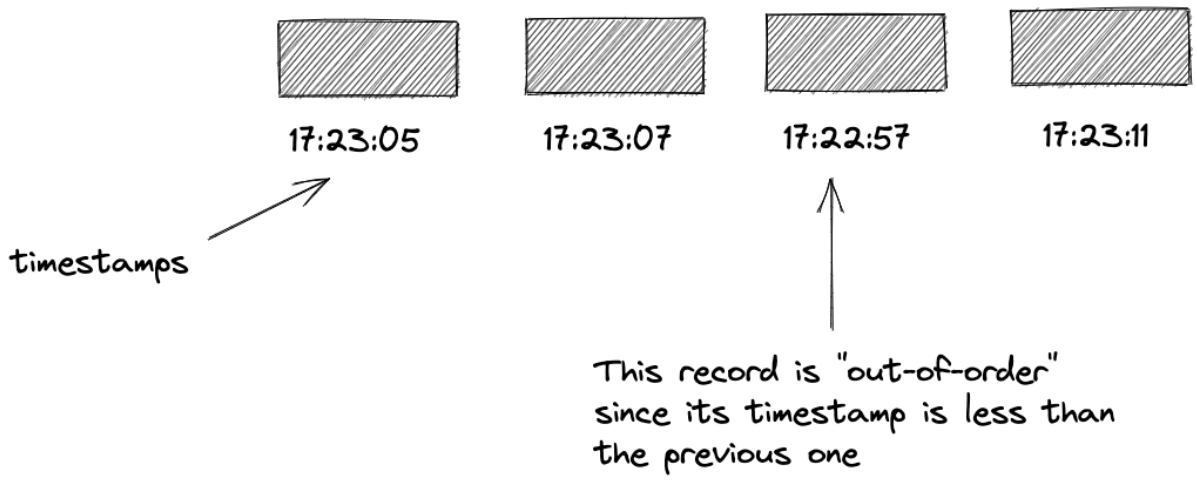


Figure 8.10 Out of order records didn't arrive in the correct sequence

So an out-of-order record is simply one where the timestamp is less than the previous one.

Now moving on to grace, it is the amount of time after a window is considered closed that you're willing to allow an out-of-order record into the aggregation. Here's an illustration demonstrating the concept:

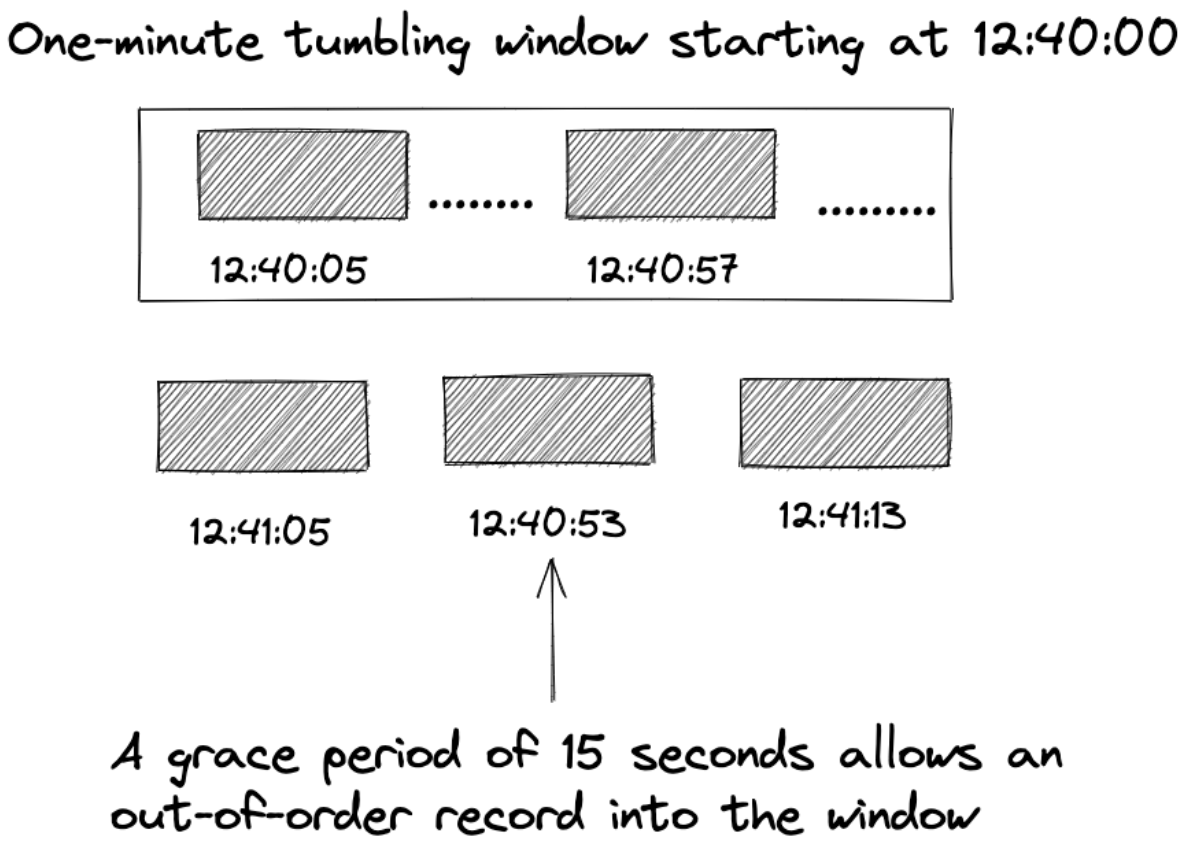


Figure 8.11 Grace is the amount of time you'll allow out-of-order records into a window after its configured close time

So from looking at the illustration, grace allows records into an aggregation that ***would have been included were they on-time***, and allows for a more accurate calculation. Once the grace period has expired, any out-of-order records are considered late and dropped. In the case of our example above, since we're not providing a grace period, when the window closes Kafka Streams drops any out-of-order records.

We'll revisit windowing and grace periods when we discuss timestamps later in the chapter, but for now it's enough to understand that timestamps on the records drive the windowing behavior and grace is a way to ensure you're getting the most accurate calculations by including records that arrive out of order.

8.12 Tumbling windows

Now let's get back to our discussion of window types and move on to the tumbling window. A tumbling window is actually a special case of a hopping window where the advance of the window is the same as its size. Since it advances the size of the window the calculation of the window contains no duplicate results. Let's take a look of an illustration showing the tumbling window in action:

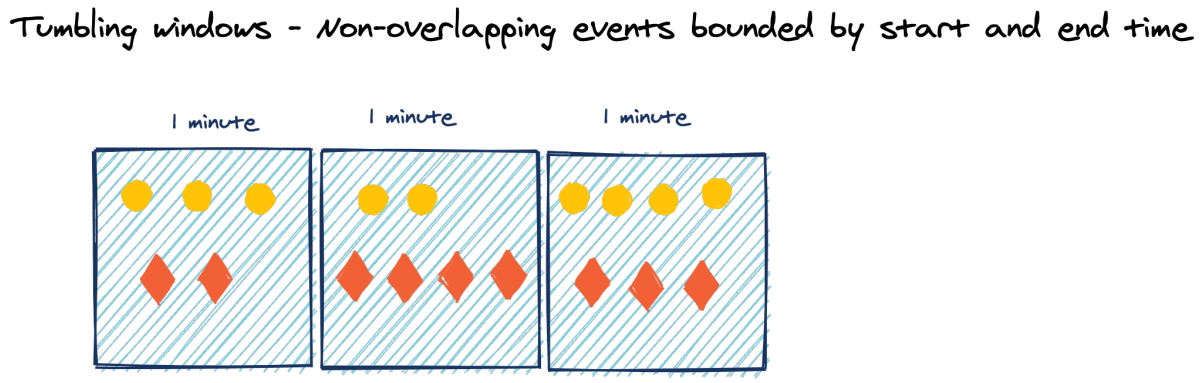


Figure 8.12 Tumbling windows move to the right by an advance equal to the size of the window

So you can see here how a tumbling window gets its name - thinking of the window as square when it's time to advance it "tumbles" into an entirely new space, and as a consequence it's guaranteed to not have any overlapping results. Specifying to use a tumbling window is easy, you simple leave off the `advanceBy` method and the size you set automatically becomes the advance time. Here's the code for setting up a tumbling window aggregation:

Listing 8.12 Setting up tumbling windows for an aggregation

```
//Some details omitted for clarity
countStream.groupByKey()
    .windowedBy(TimeWindows.ofSizeAndGrace(Duration.ofMinutes(1)   ❶
                           ,Duration.ofSeconds(30)))   ❷
    .count(Materialized.as("Tumbling-window-counting-store"))
```

❶ Setting up the tumbling window of one minute

❷ Using 30 seconds for the grace period

From looking at annotation one using a tumbling window is simply a matter not setting the advance time. Also I'd like to point out that in this example you are using a grace period of thirty seconds, so there's a second `Duration` parameter passed into the `TimeWindows.ofSizeAndGrace` method. Note that I'm only showing the required code for tumbling windows with a grace period, but the source code contains a full runnable example.

The choice of using a tumbling or a hopping window depends entirely on your use case. A hopping window gives you finer grained results with potentially overlapping results, but the tumbling window result are a little more course-grained but will not contain any overlapping records. One thing to keep in mind is that a hopping window is re-evaluated more frequently, so how often you want to observe the windowed results is one potential determinant.

8.13 Session windows

Next up in our tour of window type is the session window. The session window differs from hopping and tumbling in that it doesn't have fixed size. Instead you specify an inactivity gap; if there's no new records within the gap time, Kafka Streams closes the window. Any subsequent records coming in after the inactivity gap result in creating a new session. Otherwise it will continue to grow in size Looking at a visual pictorial of a session window is in order to fully understand how it works:

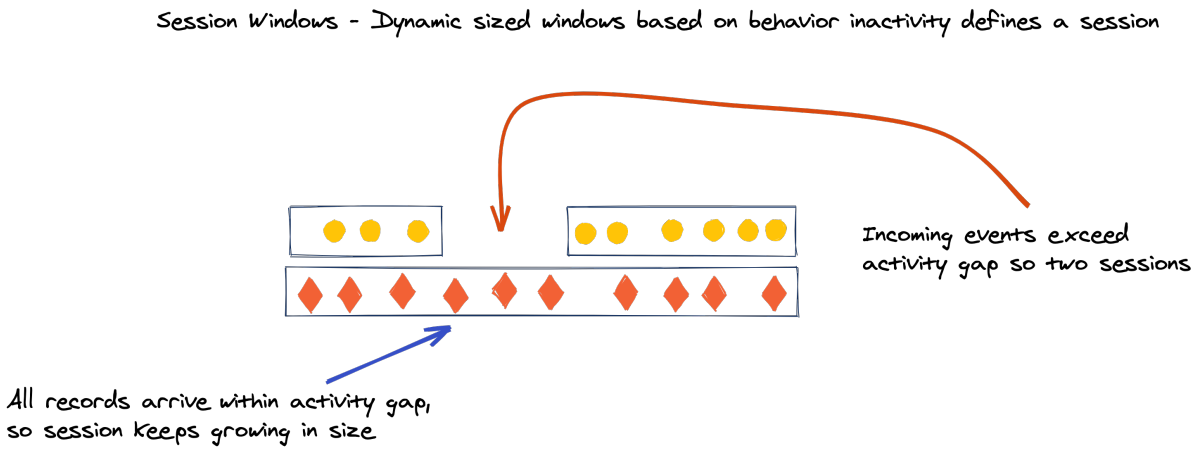


Figure 8.13 Session windows continue to grow unless no new records arrive before the inactivity gap expires

By looking at the illustration you can see that a session window is driven by behavior, unlike the hopping or tumbling window which are governed by time. Let's take a look at an example of using a session window. As before I'm only going to show the essential part here, the source code contains the full, runnable example.

Listing 8.13 Setting up the session window

```
//Some details omitted

countStream.groupByKey()
 .windowedBy(SessionWindows.ofInactivityGapAndGrace(Duration.ofMinutes(1),   ❶
                     Duration.ofSeconds(30)))    ❷
 .count(Materialized.as("Session-window-counting-store"))
```

❶ Using a session window for the aggregation

❷ Specifying a grace period

So to use sessions with your aggregation is to use a `SessionWindows` factory method. In this case you specify an inactivity period of one minute and you include a grace period as well. The grace period for session window works in the similar manner, it provides a time for Kafka Streams to include out-of-order records arriving after the inactivity period passes. As with the other window implementations, there's also a method you can use to specify no grace period.

The choice to use a session window vs. hopping/tumbling is more clear cut, it's best suited where you are tracking behavior. For example think of a user on a web application, as long as they are active on the site you'll want to get calculate the aggregation and it's impossible to know how long that could be.

8.14 Sliding windows

We're now on to the last window type to cover, `SlidingWindows`. The sliding window is a fixed-size window, but instead of specifying the size, it's the difference between timestamps that determine if a record is added to the window. So it's a combination of time-based window, but To fully understand how a `SlidingWindow` operates, take a look at the following illustration:

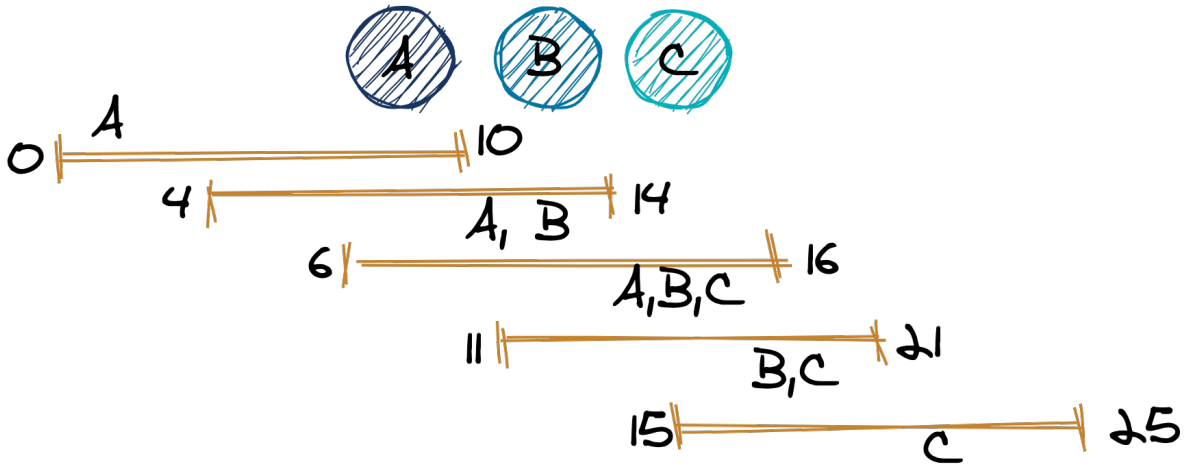


Figure 8.14 Sliding windows are fixed-size and slide along the time-axis

As you can see from the diagram, two records are in the same window if the difference in the value of their timestamps falls within the window size. So as the `SlidingWindow` slides along records may end up in several overlapping calculations, but each window will contain a unique set of records. Another way to look at the `SlidingWindow` is that as it moves along the time-axis, records come into the window and others fall out on continual basis.

Here's how you'd set up a sliding window for an aggregation:

Listing 8.14 Setting up the sliding window

```
//Some details omitted

 countStream.groupByKey()
    .windowedBy(SlidingWindows.ofTimeDifferenceWithNoGrace(
            Duration.ofSeconds(30)))   ❶
    .count(Materialized.as("Sliding-window-counting-store"))
```

❶ Specifying a sliding window with time difference of 30 seconds

As with the all the windowed options we've seen so far, it's simply a matter of providing a factory method for the desired windowing functionality. In this case, you've set the time difference to thirty seconds with no grace period. Just like the other windowing options, you could specify a grace period as well with the `SlidingWindows.ofTimeDifferenceWithGrace` method.

The determining factor to go with a sliding window over a hopping or tumbling window is a matter of how fine-grained of a calculation is desired. When you need to generate a continual running average or sum is a great use-case for the `SlidingWindow`.

You could achieve similar behavior with a `HoppingWindow` by using a small advance interval. But this approach will result in poor performance because the hopping windows will create redundant windows and performing aggregation operations over them is inefficient. Compared to the `SlidingWindow` that only creates windows containing distinct items so the calculations are more efficient.

This wraps up our coverage of the different windowing types that Kafka Streams provides, but before we move on to another section we will cover one more feature that is available for all windows. As records flow into the windowed aggregation processor, Kafka Streams continually updates the aggregation with the new records. Kafka Streams updates the `KTable` with the new aggregation, and it forwards the previous aggregation results to downstream operators.

Remember that Kafka Streams uses caching for stateful operations, so every update doesn't flow downstream. It's only on cache flush or a commit that the updates make it downstream. But depending on the size of your window, this means that you'll get partial results of your windowing operations until the window closes. In many cases receiving a constant flow of fresh calculations is desired.

But in some cases, you may want to have a single, **final** result forwarded downstream from a windowed aggregation. For example, consider a case where your application is tracking IoT sensor readings with a count of temperature readings that exceed a given threshold over the past 30 minutes. If you find a temperature breach, you'll want to send an alert. But with regular updates, you'll have to provide extra logic to determine if the result is an intermediate or final one. Kafka Streams provides an elegant solution to this situation, the ability to suppress intermediate results.

8.15 Suppression

For stateless operations the behavior of always forwarding a result is expected in the nature of a stream processing system. But sometimes for a windowed operation it's desirable for a final result when the window closes. For example, take the case of the tumbling window example above, instead of incremental results, you want a single final count.

NOTE **Final results are only available for windowed operations. With an event streaming application like Kafka Streams, the number of incoming records is infinite, so there's never a point we can consider a record final. But since a windowed aggregation represents a discrete point in time, the available record when the window closes can be considered final.**

So far you've learned about the different windowing operations available, but they all yield intermediate results, now let's suppose you only want the final result from the window. For that you'd use the `KTable.suppress` operation.

The `KTable.suppress` method takes a `Suppressed` configuration object which allows you to configure the suppression in two ways:

1. Strict - results are buffered by time and the buffering is strictly enforced by never emitting a result early until the time bound is met
2. Eager - results are buffered by size (number of bytes) or by number of records and when these conditions are met, results are emitted downstream. This will reduce the number of downstream results, but doesn't guarantee a final one.

So you have two choices - the strict approach which guarantees a final result or the eager one which could produce a final result, but also has the likelihood of emitting a few intermediate results as well. The trade-off to make can be thought of this way - with strict buffering, the size of the buffer doesn't have any bounds, so the possibility of getting an `OutOfMemory` (OOM) exists, but with eager buffering you'll never hit an OOM exception, but you could end up with multiple results. While the possibility of incurring an OOM may sound extreme, if you have feel the buffer won't get that large or you have a sufficiently large heap available then using the strict configuration should be OK.

NOTE **The possibility of an OOM is not as harsh as it seems at first glance. All Java applications that use a data-structures in-memory, List, Set or Map have the potential for causing an OOM if you continually add to them. To use them effectively requires a balance of knowledge between the incoming data and the amount of heap you have available.**

Let's take a look now at how at an example of using suppression.

Listing 8.15 Setting up suppression on a KStream aggregation

```
countStream.groupByKey()
    .windowedBy(TimeWindows.ofSizeWithNoGrace(Duration.ofMinutes(1)))   ❶
    .count(Materialized.as("Tumbling-window-suppressed-counting-store"))
    .suppress(untilWindowCloses(unbounded()))   ❷
```

❶ Creating a one-minute tumbling window

❷ Suppressing results until the window closes with an unbounded configuration

So setting up suppression is as easy as adding one line of code, which you can see at annotation two. In this case you're suppressing all output until the window closes along with an unbounded buffering of records. For testing scenarios this is an acceptable configuration, but if running with such a configuration in a production setting gives you pause, let's quick show two alternative settings.

First you can configure the final result with a maximum number records or bytes, then if the constraint is violated, you can have a graceful shutdown:

Listing 8.16 Setting up suppression for final result controlling the potential shutdown

```
.suppress(untilWindowCloses(maxRecords(10_000)    ❶
                            .shutDownWhenFull())  ❷
```

❶ Setting max records to 10K

❷ Specifying to shutdown if limit is reached

Here you're specifying to go with an unbounded window, but you'd rather have a graceful shutdown should the buffer start to grow beyond what you feel is a reasonable amount. So in this case you specify the maximum number of records is 10K and should the buffering exceed that number, the application will shut down gracefully.

Note that we technically could have used a `shutDownWhenFull` with our original suppression example, but the default limit is `LONG.MAX_VALUE`, so in practice most likely that you'd get an OOM exception before reaching that size constraint. With this change you're favoring shutting down before emitting a possible non-final result.

On the other hand, if you'd rather trade-off a possible non-final result over shutting down you could use a configuration like this:

Listing 8.17 Using suppression emulating a final result with a possible early result instead of shutting down

```
.suppress(untilTimeLimit(Duration.ofMinutes(1),    ❶
                         maxRecords(1000)          ❷
                        .emitEarlyWhenFull()))     ❸
```

❶ Setting time limit of one hour before sending result downstream

❷ Specifying to buffer a maximum of 1K records

❸ Take the action of emitting a record if the maximum number of buffered records is reached

With this example, you've set the time limit to match the size of the window (plus any grace

period) so you're reasonable sure to get a final result, but you've set the maximum size of the buffer, and if the number of records reaches that size, the processor will forward a record regardless if the time limit is reached or not. One thing to bear in mind is if you want to set the time limit to correspond to the window closing, you need to include the grace period, if any, as well in the time limit.

This wraps up our discussion on suppression of aggregations in Kafka Steams. Even though the examples in the suppression section only demonstrated using the `KStream` and windowed aggregations, you could apply the same principal to non-windowed `KTable` aggregations by using the time-limit API of suppression.

Now let's move on to the last section of this chapter, timestamps in Kafka Streams.

8.16 Timestamps in Kafka Streams

Earlier in the book, we discussed timestamps in Kafka records. In this section, we'll discuss the use of timestamps in Kafka Streams. Timestamps play a role in key areas of Kafka Streams functionality:

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

With stream processing in general, you can group timestamps into three categories, as shown in figure 8.10:

- *Event time* — A timestamp set when the event occurred, usually embedded in the object used to represent the event. For our purposes, we'll consider the timestamp set when the `ProducerRecord` is created as the event time as well.
- *Ingestion time* — A timestamp set when the data first enters the data processing pipeline. You can consider the timestamp set by the Kafka broker (assuming a configuration setting of `LogAppendTime`) to be ingestion time.
- *Processing time* — A timestamp set when the data or event record first starts to flow through a processing pipeline.

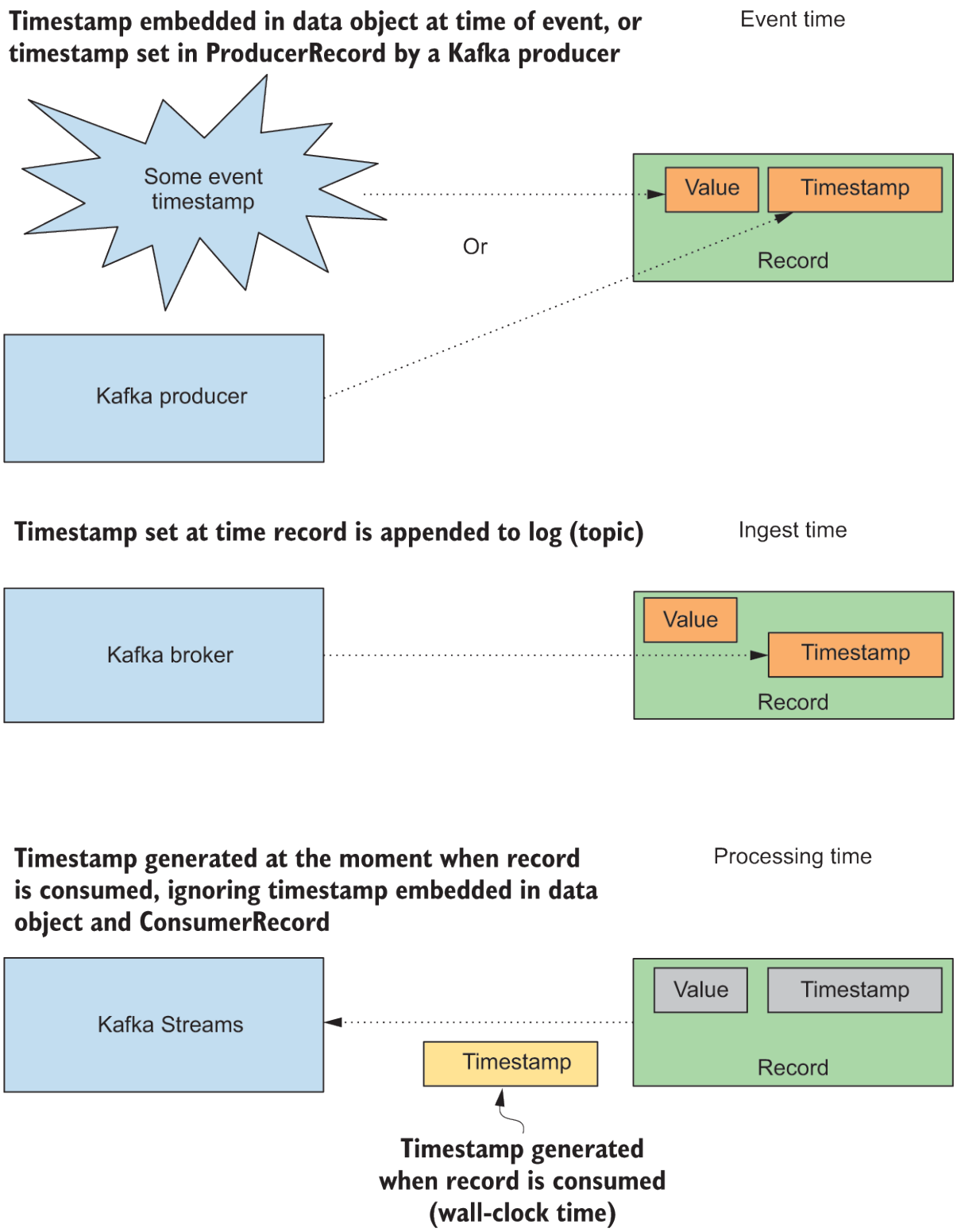


Figure 8.15 There are three categories of timestamps in Kafka Streams: event time, ingestion time, and processing time.

You'll see in this section how the Kafka Streams by using a `TimestampExtractor`, gives you the ability to chose which timestamp semantics you want to support.

> **NOTE** So far, we've had an implicit assumption that clients and brokers are located in the same time zone, but that might not always be the case. When using timestamps, it's safest to normalize the times using the UTC time zone, eliminating any confusion over which brokers and clients are using which time zones.

In most cases using event-time semantics, the timestamp placed in the metadata by the `ProducerRecord` is sufficient. But there may be cases when you have different needs. Consider these examples:

- You're sending messages to Kafka with events that have timestamps recorded in the message objects. There's some lag time in when these event objects are made available to the Kafka producer, so you want to consider only the embedded timestamp.
- You want to consider the time when your Kafka Streams application processes records as opposed to using the timestamps of the records.

8.17 The TimestampExtractor

To enable different processing semantics, Kafka Stream provides a `TimestampExtractor` interface with one abstract and four concrete implementations. If you need to work with timestamps embedded in the record values, you'll need to create a custom `TimestampExtractor` implementation. Let's briefly look at the included implementations and implement a custom `TimestampExtractor`.

Almost all of the provided `TimestampExtractor` implementations work with timestamps set by the producer or broker in the message metadata, thus providing either event-time processing semantics (timestamp set by the producer) or log-append-time processing semantics (timestamp set by the broker). Figure 4.19 demonstrates pulling the timestamp from the `ConsumerRecord` object.

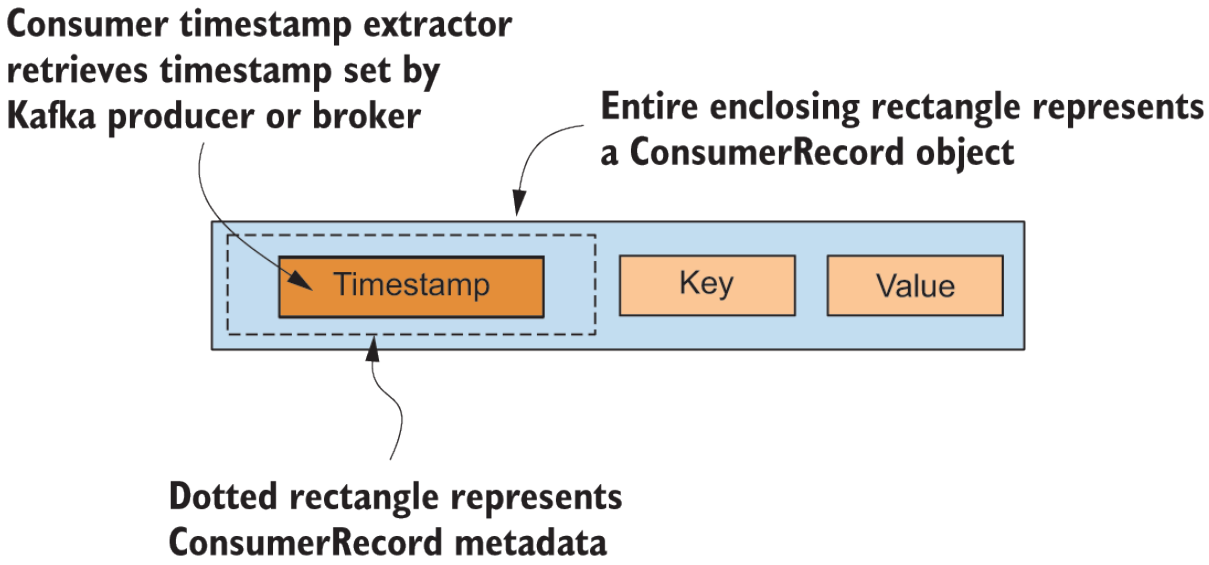


Figure 8.16 Timestamps in the `ConsumerRecord` object: either the producer or broker set this timestamp, depending on your configuration.

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

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

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

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

8.18 WallclockTimestampExtractor

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

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

version.

8.19 Custom TimestampExtractor

To work with timestamps (or calculate one) in the value object from the `ConsumerRecord`, you'll need a custom extractor that implements the `TimestampExtractor` interface. For example, let's say you are working with IoT sensors and part of the information is the exact time of the sensor reading. It's important for your calculations to have the precise timestamp, so you'll want to use the one embedded in the record sent to Kafka and not the one set by the producer.

The figure here depicts using the timestamp embedded in the value object versus one set by Kafka (either producer or broker).

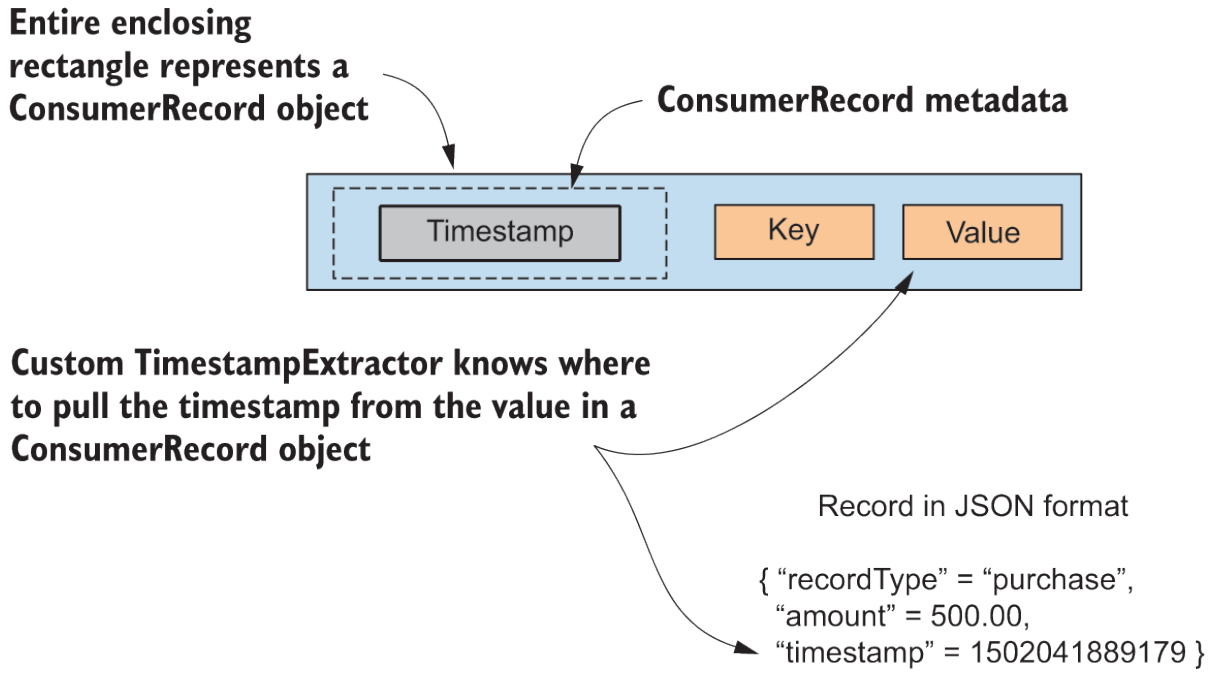


Figure 8.17 A custom `TimestampExtractor` provides a timestamp based on the value contained in the `ConsumerRecord`. This timestamp could be an existing value or one calculated from properties contained in the value object.

Here's an example of a `TimestampExtractor` implementation (found in src/main/java/bbejeck/chapter_4/timestamp_extractor/TransactionTimestampExtractor.java), also used in the join example from listing 4.12 in the section "Implementing the Join" (although not shown in the text, because it's a configuration parameter).

Listing 8.18 Custom `TimestampExtractor`

```
public class TransactionTimestampExtractor implements TimestampExtractor {

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

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

❷ Returns the timestamp recorded at the point of sale

In the join example, you used a custom `TimestampExtractor` because you wanted to use the timestamps of the actual purchase time. This approach allows you to join the records even if there are delays in delivery or out-of-order arrivals.

8.20 Specifying a TimestampExtractor

Now that we've discussed how timestamp extractors work, let's tell the application which one to use. You have two choices for specifying timestamp extractors.

The first option is to set a global timestamp extractor, specified in the properties when setting up your Kafka Streams application. If no property is set, the default setting is `FailOnInvalidTimestamp.class`. For example, the following code would configure the `TransactionTimestampExtractor` via properties when setting up the application:

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

The second option is to provide a `TimestampExtractor` instance via a `Consumed` object:

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

The advantage of doing this is that you have one `TimestampExtractor` per input source, whereas the other option provides a `TimestampExtractor` instance used application wide.

8.21 Streamtime

Before we end this chapter, we should discuss how Kafka Streams keeps track of time while processing, that is by using streamtime. Streamtime is not another category of timestamp, it is the current time in a Kafka Streams processor. As Kafka Streams selects the next record to process by timestamp and as processing continues the values will increase. Streamtime is the largest timestamp seen by a processor and represents the current time for it. Since a Kafka Streams application is broken down into tasks and a task is responsible for records from a given partition, the value of streamtime is not global in a Kafka Streams application, it's only unique at the task level.

Streamtime only moves forward never backwards. Out of order records are always processed, with the exception of windowed operations depending on the grace period, but its timestamp does not affect streamtime. Here's an illustration showing how streamtime works in a Kafka Streams application.

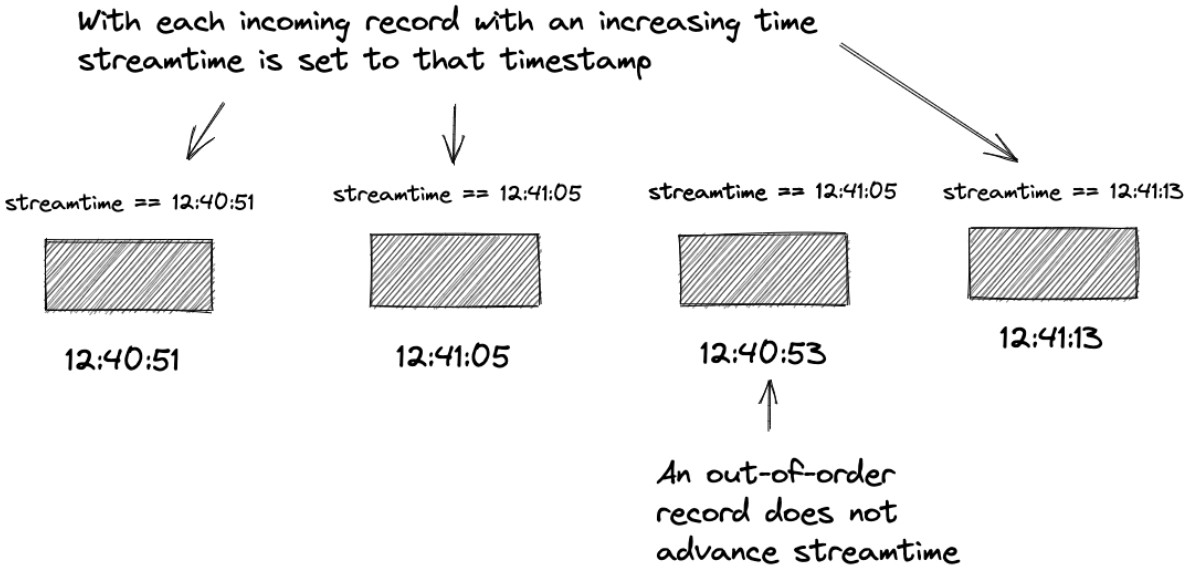


Figure 8.18 Streamtime represents the highest timestamp seen so far and is the current time of the application

So as the illustration shows, the current time of the application moves forward as records go through the topology and out of order records still go through the application but do not change streamtime.

Streamtime is vital for the correctness of windowed operations as a window only advances and closes as streamtime moves forward. If the source topics for your application are bursty or have a sporadic sustain volume of records, you might encounter a situation where you don't observe windowed results. This apparent lack of processing is due to the fact that there hasn't been enough incoming records to move streamtime forward to force window calculations.

This effect that timestamps have on operations in Kafka Streams is important to keep in mind when testing applications, as manually adjusting the value of timestamps can help you drive useful tests to validate behavior. We'll talk more about using timestamps for testing in the chapter on testing.

Streamtime also comes into play when you have punctuations which we'll cover in the next chapter when we discuss the Processor API.

8.22 Summary

- The `KTable` is an update stream and models a database table where the primary key is the key from the key-value pair in the stream. Records with the same key are considered updates to previous ones with the same key. Aggregations with the `KTable` are analogous to running a `Select…. Group By` SQL query against a relational database table.
- Performing joins with a `KStream` against a `KTable` is a great way to enrich an event stream. The `KStream` contains the event data and the `KTable` contains the facts or dimension data.
- It's possible to perform joins between two `KTable`s and you can also do a foreign key join between two `KTable`s
- The `GlobalKTable` contains all records of the underlying topic as it's not sharded so each application instance contains all the records making it suitable for acting as a reference table. Joins with the `GlobalKTable` don't require co-partitioning with the `KStream`, you can supply a function that calculates the correct key for the join.
- Windowing is a way to calculate aggregations for a given period of time. Like all other operations in Kafka Streams, new incoming records mean an update is released downstream, but windowed operations can use suppression to only have a single final result when the window closes.
- There are four types of windows hopping, tumbling, sliding, and session. Hopping and tumbling windows are fixed in size by time. Sliding windows are fixed in size by time, but record behavior drives record inclusion in a window. Session windows are completely driven by record behavior, and the window can continue to grow as long as incoming records are within the inactivity gap.
- Timestamps drive the behavior in a Kafka Streams application and this most obvious in windowed operations as the timestamps of the records drive the opening and closing of these operations. Streamtime is the highest timestamp viewed by a Kafka Streams application during it's processing.
- Kafka Streams provides different `TimestampExtractor` instances so you can use different timestamp semantics event-time, log-append-time, or processing time in your application.