Stateful Processing

In the previous chapter, we learned how to perform stateless transformations of record streams using the `KStream` abstraction and a rich set of stateless operators that are available in Kafka Streams. Since stateless transformations don't require any memory of previously seen events, they are easy to reason about and use. We treat every event as an immutable *fact* and process it independently of other events.

However, Kafka Streams also gives us the ability to capture and remember information about the events we consume. The captured information, or *state*, allows us to perform more advanced stream processing operations, including joining and aggregating data. In this chapter, we will explore stateful stream processing in detail. Some of the topics we will cover include:

- The benefits of stateful stream processing
- The differences between facts and behaviors
- What kinds of stateful operators are available in Kafka Streams
- How state is captured and queried in Kafka Streams
- How the `KTable` abstraction can be used to represent local, partitioned state
- How the `GlobalKTable` abstraction can be used to represent global, replicated state
- How to perform stateful operations, including joining and aggregating data
- How to use interactive queries to expose state

As with the previous chapter, we will explore these concepts using a tutorial-based approach. This chapter's tutorial is inspired by the video game industry, and we'll be building a real-time leaderboard that will require us to use many of Kafka Streams' stateful operators. Furthermore, we'll spend a lot of time discussing joins since this is

one of the most common forms of data enrichment in stateful applications. But before we get to the tutorial, let's start by looking at some of the benefits of stateful processing.

Benefits of Stateful Processing

Stateful processing helps us understand the *relationships between events* and leverage these relationships for more advanced stream processing use cases. When we are able to understand how an event relates to other events, we can:

- Recognize patterns and behaviors in our event streams
- Perform aggregations
- Enrich data in more sophisticated ways using joins

Another benefit of stateful stream processing is that it gives us an additional abstraction for representing data. By replaying an event stream one event at a time, and saving the latest state of each key in an embedded key-value store, we can build a point-in-time representation of continuous and unbounded record streams. Thesepoint-in-time representations, or snapshots, are referred to as *tables*, and KafkaStreams includes different types of table abstractions that we'll learn about in this chapter.

Tables are not only at the heart of stateful stream processing, but when they are materialized, they can also be queried. This ability to query a real-time snapshot of a fast-moving event stream is what makes Kafka Streams a *stream-relational* processing platform,[1] and enables us to not only build stream processing applications, but also low-latency, event-driven microservices as well.

Finally, stateful stream processing allows us to understand our data using more sophisticated mental models. One particularly interesting view comes from NeilAvery, who discusses the differences between facts and behaviors in his discussion ofevent-first thinking:

> An event represents a fact, something happened; it is immutable…

Stateless applications, like the ones we discussed in the previous chapter, are fact-driven. Each event is treated as an independent and atomic fact, which can beprocessed using immutable semantics (think of inserts in a never-ending stream), and then subsequently forgotten.

However, in addition to leveraging stateless operators to filter, branch, merge, and transform facts, we can ask even more advanced questions of our data if we learn how to model *behaviors* using stateful operators. So what are behaviors? According to Neil:

> The accumulation of facts captures behavior.

You see, events (or facts) rarely occur in isolation in the real world. Everything is interconnected, and by capturing and remembering facts, we can begin to understand

their meaning. This is possible by understanding events in their larger historical context, or by looking at other, related events that have been captured and stored by our application.

A popular example is shopping cart abandonment, which is a behavior comprised of multiple facts: a user adds one or more items to a shopping cart, and then a session is terminated either manually (e.g., the user logs off) or automatically (e.g., due to a long period of inactivity). Processing either fact independently tells us very little about where the user is in the checkout process. However, collecting, remembering, and analyzing each of the facts (which is what stateful processing enables) allows us to recognize and react to the *behavior*, and provides much greater business value than viewing the world as a series of unrelated events.

Now that we understand the benefits of stateful stream processing, and the differ- ences between facts and behaviors, let's get a preview of the stateful operators in Kafka Streams.

Preview of Stateful Operators

Kafka Streams includes several stateful operators that we can use in our processor topologies. Table 4-1 includes an overview of several operators that we will be working with in this book.

Table 4-1. Stateful operators and their purpose

Use case	Purpose	Operators
Joining data	Enrich an event with additional information or context that was captured in a separate stream or table	• `join` (inner join) • `leftJoin` • `outerJoin`
Aggregating data	Compute a continuously updating mathematical or combinatorial transformation of related events	• `aggregate` • `count` • `reduce`
Windowing data	Group events that have close temporal proximity	• `windowedBy`

Furthermore, we can *combine* stateful operators in Kafka Streams to understand even more complex relationships/behaviors between events. For example, performing a *windowed join* allows us to understand how discrete event streams relate during a certain period of time. As we'll see in the next chapter, *windowed aggregations* are another useful way of combining stateful operators.

Now, compared to the stateless operators we encountered in the previous chapter, stateful operators are more complex under the hood and have additional compute and storage[2] requirements. For this reason, we will spend some time learning about the inner workings of stateful processing in Kafka Streams before we start using the stateful operators listed in Table 4-1.

Perhaps the most important place to begin is by looking at how state is stored and queried in Kafka Streams.

State Stores

We have already established that stateful operations require our application to maintain some memory of previously seen events. For example, an application that counts the number of error logs it sees needs to keep track of a single number for each key: a rolling count that gets updated whenever a new error log is consumed. This count represents the historical context of a record and, along with the record key, becomes part of the application's state.

To support stateful operations, we need a way of storing and retrieving the remembered data, or state, required by each stateful operator in our application (e.g., `count`, `aggregate`, `join`, etc.). The storage abstraction that addresses these needs in Kafka Streams is called a *state store*, and since a single Kafka Streams application can leverage *many* stateful operators, a single application may contain several state stores.

There are many state store implementations and configuration possibilities available in Kafka Streams, each with specific advantages, trade-offs, and use cases. Whenever you use a stateful operator in your Kafka Streams application, it's helpful to consider which type of state store is needed by the operator, and also how to configure the state store based on your optimization criteria (e.g., are you optimizing for high throughput, operational simplicity, fast recovery times in the event of failure, etc.). In most cases, Kafka Streams will choose a sensible default if you don't explicitly specify a state store type or override a state store's configuration properties.

Since the variation in state store types and configurations makes this quite a deep topic, we will initially focus our discussion on the common characteristics of all of thedefault state store implementations, and then take a look at the two broad categories of state stores: persistent and in-memory stores.

Common Characteristics

The default state store implementations included in Kafka Streams share some common properties. We will discuss these commonalities in this section to get a better idea of how state stores work.

Embedded

The default state store implementations that are included in Kafka Streams are *embedded* within your Kafka Streams application at the task level. The advantage of embedded state stores, as opposed to using an external storage engine, is that the latter would require a network call whenever state needed to be accessed, and would therefore introduce unnecessary latency and processing bottlenecks. Furthermore, since state stores are embedded at the task level, a whole class of concurrency issues for accessing shared state are eliminated.

Additionally, if state stores were remote, you'd have to worry about the availability of the remote system separately from your Kafka Streams application. Allowing Kafka Streams to manage a local state store ensures it will always be available and reduces the error surface quite a bit. A *centralized* remote store would be even worse, since it would become a single point of failure for all of your application instances. Therefore, Kafka Streams' strategy of colocating an application's state alongside the application itself not only improves performance (as discussed in the previous paragraph), but also availability.

All of the default state stores leverage RocksDB under the hood. RocksDB is a fast, embedded key-value store that was originally developed at Facebook. Since it supports arbitrary byte streams for storing key-value pairs, it works well with Kafka, which also decouples serialization from storage. Furthermore, both reads and writes are extremely fast, thanks to a rich set of optimizations that were made to the forked LevelDB code.

Multiple access modes

State stores support multiple access modes and query patterns. Processor topologies require read and write access to state stores. However, when building microservices using Kafka Streams' *interactive queries* feature, clients require only *read* access to the underlying state. This ensures that state is never mutable outside of the processor topology, and is accomplished through a dedicated read-only wrapper that clients can use to safely query the state of a Kafka Streams application.

Fault tolerant

By default, state stores are backed by changelog topics in Kafka. In the event of failure, state stores can be restored by replaying the individual events from the underlying changelog topic to reconstruct the state of an application. Furthermore, Kafka Streams allows users to enable standby replicas for reducing the amount of time it takes to rebuild an application's state. These standby replicas (sometimes called *shadow copies*) make state stores *redundant*, which is an important characteristic of highly available systems. In addition, applications that allow their state to be queried can rely

on standby replicas to serve query traffic when other application instances godown, which also contributes to high availability.

Key-based

Operations that leverage state stores are key-based. A record's key defines the relationship between the current event and other events. The underlying data structure will vary depending on the type of state store you decide to use, but each implementation can be conceptualized as some form of key-value store, where keys may be simple, or even compounded (i.e., multidimensional) in some cases.

Now that we understand the commonalities between the default state stores in Kafka Streams, let's look at two broad categories of state stores to understand the differences between certain implementations.

Persistent Versus In-Memory Stores

One of the most important differentiators between various state store implementations is whether or not the state store is *persistent*, or if it simply stores remembered information *in-memory* (RAM). Persistent state stores flush state to disk asynchronously (to a configurable *state directory*), which has two primary benefits:

- State can exceed the size of available memory.
- In the event of failure, persistent stores can be restored quicker than in-memory stores.

To clarify the first point, a persistent state store may keep some of its state in- memory, while writing to disk when the size of the state gets too big (this is called *spilling to disk*) or when the write buffer exceeds a configured value. Second, since theapplication state is persisted to disk, Kafka Streams does not need to replay the entiretopic to rebuild the state store whenever the state is lost (e.g., due to system failure, instance migration, etc.). It just needs to replay whatever data is missing between the time the application went down and when it came back up.

The state store directory used for persistent stores can be set using the `StreamsConfig.STATE_DIR_CONFIG` property. The default location is */tmp/kafka-streams*, but it is highly recommended that you override this to a directory outside of */tmp*.

The downside is that persistent state stores are operationally more complex and can be slower than a pure in-memory store, which always pulls data from RAM. The additional operational complexity comes from the secondary storage requirement (i.e., disk-based storage) and, if you need to tune the state store, understanding RocksDB and its configurations (the latter may not be an issue for most applications).

Regarding the performance gains of an in-memory state store, these may not be drastic enough to warrant their use (since failure recovery takes longer). Adding more partitions to parallelize work is always an option if you need to squeeze more perfor-

mance out of your application. Therefore, my recommendation is to start with persistent stores and only switch to in-memory stores if you have measured a noticeable performance improvement and, when quick recovery is concerned (e.g., in the event your application state is lost), you are using standby replicas to reduce recovery time.

Now that we have some understanding of what state stores are, and how they enable stateful/behavior-driven processing, let's take a look at this chapter's tutorial and see some of these ideas in action.

Introducing Our Tutorial: Video Game Leaderboard

In this chapter, we will learn about *stateful processing* by implementing a video game leaderboard with Kafka Streams. The video game industry is a prime example of where stream processing excels, since both gamers and game systems require low- latency processing and immediate feedback. This is one reason why companies like Activision (the company behind games like *Call of Duty* and remasters of *Crash Ban-dicoot* and *Spyro*) use Kafka Streams for processing video game telemetry.

The leaderboard we will be building will require us to model data in ways that we haven't explored yet. Specifically, we'll be looking at how to use Kafka Streams' table abstractions to model data as a sequence of updates. Then, we'll dive into the topics of joining and aggregating data, which are useful whenever you need to understand or compute the relationship between multiple events. This knowledge will help you solve more complicated business problems with Kafka Streams.

Once we've created our real-time leaderboard using a new set of stateful operators, we will demonstrate how to query Kafka Streams for the latest leaderboard information using *interactive queries*. Our discussion of this feature will teach you how to build event-driven microservices with Kafka Streams, which in turn will broaden the type of clients we can share data with from our stream processing applications.

Without further ado, let's take a look at the architecture of our video game leaderboard. Figure 4-1 shows the topology design we'll be implementing in this chapter. Additional information about each step is included after the diagram.

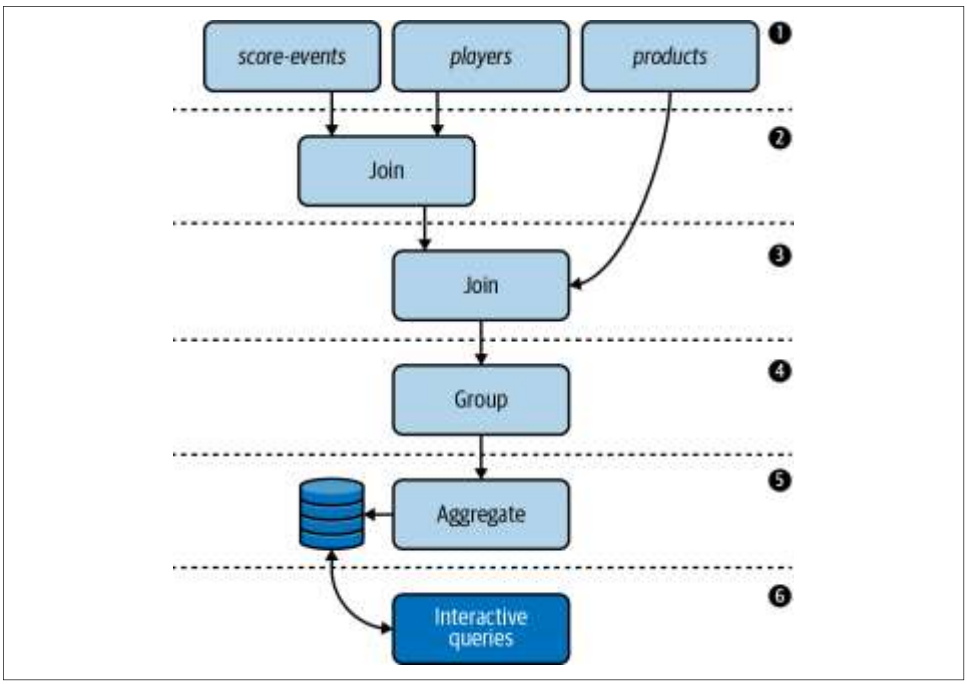

Figure 4-1. The topology that we will be implementing in our stateful video game leaderboard application

❶ Our Kafka cluster contains three topics:

- The `score-events` topic contains game scores. The records are unkeyed and are therefore distributed in a round-robin fashion across the topic's partitions.
- The `players` topic contains player profiles. Each record is keyed by a player ID.
- The `products` topic contains product information for various video games. Each record is keyed by a product ID.

❷ We need to enrich our score events data with detailed player information. We can accomplish this using a join.

❸ Once we've enriched the `score-events` data with player data, we need to add detailed product information to the resulting stream. This can also be accomplished using a join.

❹ Since grouping data is a prerequisite for aggregating, we need to group the enriched stream.

❺ We need to calculate the top three high scores for each game. We can use Kafka Streams' aggregation operators for this purpose.

❻ Finally, we need to expose the high scores for each game externally. We will accomplish this by building a RESTful microservice using the *interactive queries*

feature in Kafka Streams.

With our topology design in hand, we can now move on to the project setup.

Project Setup

The code for this chapter is located at *https://github.com/mitch-seymour/mastering-kafka-streams-and-ksqldb.git*.

If you would like to reference the code as we work our way through each topology step, clone the repo and change to the directory containing this chapter's tutorial. The following command will do the trick:

```
$ git clone git@github.com:mitch-seymour/mastering-kafka-streams-and-ksqldb.git
$ cd mastering-kafka-streams-and-ksqldb/chapter-04/video-game-leaderboard
```

You can build the project anytime by running the following command:

```
$ ./gradlew build --info
```

Now that our project is set up, let's start implementing our video game leaderboard.

Data Models

As always, we'll start by defining our data models. Since the source topics contain JSON data, we will define our data models using POJO data classes, which we will serialize and deserialize using our JSON serialization library of choice (throughout this book, we use Gson, but you could easily use Jackson or another library).

I like to group my data models in a dedicated package in my project, for example, `com.magicalpipelines.model`. A filesystem view of where the data classes for this tutorial are located is shown here:

```
src/
└── main
    └── java
        └── com
            └── magicalpipelines
                └── model
                    ├── ScoreEvent.java ❶
                    ├── Player.java ❷
                    └── Product.java ❸
```

❶ The `ScoreEvent.java` data class will be used to represent records in the `score-events` topic.

❷ The `Player.java` data class will be used to represent records in the `players` topic.

❸ The `Product.java` data class will be used to represent records in the `products` topic.

Now that we know which data classes we need to implement, let's create a data class for each topic. Table 4-2 shows the resulting POJOs that we have implemented for this

tutorial.

Table 4-2. Example records and data classes for each topic

Kafka topic	Example record	Data class

score-events	{
	"score": 422,
	"product_id": 6,
	"player_id": 1
	}
		public class ScoreEvent {
		private Long playerId;
		private Long productId;
		private Double score;
		}

players	{
	"id": 2,
	"name": "Mitch"
	}
		public class Player {
		private Long id;
		private String name;
		}

products	{
	"id": 1,
	"name": "Super Smash Bros"
	}
		public class Product {
		private Long id;
		private String name;
		}

Adding the Source Processors

Once we've defined our data classes, we can set up our source processors. In this topology, we need three of them since we will be reading from three source topics. The first thing we need to do when adding a source processor is determine which Kafka Streams abstraction we should use for representing the data in the underlying topic.

Up until now, we have only worked with the `KStream` abstraction, which is used to represent stateless record streams. However, our topology requires us to use both the `products` and `players` topics as lookups, so this is a good indication that a table-like abstraction may be appropriate for these topics.[10] Before we start mapping our topics to Kafka Streams abstractions, let's first review the difference between `KStream`, `KTable`, and `GlobalKTable` representations of a Kafka topic. As we review each abstraction, we'll fill in the appropriate abstraction for each topic in Table 4-3.

Table 4-3. The topic-abstraction mapping that we'll update in the following sections

Kafka topic	Abstraction
`score-events`	???
`players`	???
`products`	???

KStream

When deciding which abstraction to use, it helps to determine the nature of the topic, the topic configuration, and the keyspace of records in the underlying source topic. Even though stateful Kafka Streams applications use one or more table abstractions, it is also very common to use stateless `KStreams` alongside a `KTable` or `GlobalKTable` when the mutable table semantics aren't needed for one or more data sources.

In this tutorial, our `score-events` topic contains raw score events, which are unkeyed (and therefore, distributed in a round-robin fashion) in an uncompacted topic. Since tables are key-based, this is a strong indication that we should be using a `KStream` for our unkeyed `score-events` topic. We could change our keying strategy upstream (i.e., in whatever application is producing to our source topic), but that's not always possible. Furthermore, our application cares about the *highest score* for each player, not the *latest score*, so table semantics (i.e., retain only the most recent record for a given key) don't translate well for how we intend to use the `score-events` topic, even if it were keyed.

Therefore, we will use a `KStream` for the `score-events` topic, so let's update the table in Table 4-3 to reflect this decision as follows:

Kafka topic	Abstraction
score-events	KStream
players	???
products	???

The remaining two topics, `players` and `products`, are keyed, and we only care about the latest record for each unique key in the topic. Hence, the `KStream` abstraction isn't ideal for these topics. So, let's move on and see if the `KTable` abstraction is appropriate for either of these topics.

KTable

The `players` topic is a compacted topic that contains player profiles, and each record is keyed by the player ID. Since we only care about the latest state of a player, it makes sense to represent this topic using a table-based abstraction (either `KTable` or `GlobalKTable`).

One important thing to look at when deciding between using a `KTable` or `GlobalKTable` is the keyspace. If the keyspace is very large (i.e., has high cardinality/lots of unique keys), or is expected to grow into a very large keyspace, then it makes more sense to use a `KTable` so that you can distribute fragments of the entire state across all of your running application instances. By partitioning the state in this way, we can lower the local storage overhead for each individual Kafka Streams instance.

Perhaps a more important consideration when choosing between a `KTable` or `GlobalKTable` is whether or not you need time synchronized processing. A `KTable` is time synchronized, so when Kafka Streams is reading from multiple sources (e.g., in the case of a join), it will look at the timestamp to determine which record to process next. This means a join will reflect what the combined record would have been at a certain time, and this makes the join behavior more predictable. On the other hand, `GlobalKTables` are *not* time synchronized, and are "completely populated before any processing is done." Therefore, joins are always made against the most up-to-date version of a `GlobalKTable`, which changes the semantics of the program.

In this case, we're not going to focus too much on the second consideration since we've reserved the next chapter for our discussion of time and the role it plays in

Kafka Streams. With regards to the keyspace, `players` contains a record for each unique player in our system. While this may be small depending on where we are in the life cycle of our company or product, it is a number we expect to grow significantly over time, so we'll use a `KTable` abstraction for this topic.

Figure 4-2 shows how using a `KTable` leads to the underlying state being distributed across multiple running application instances.

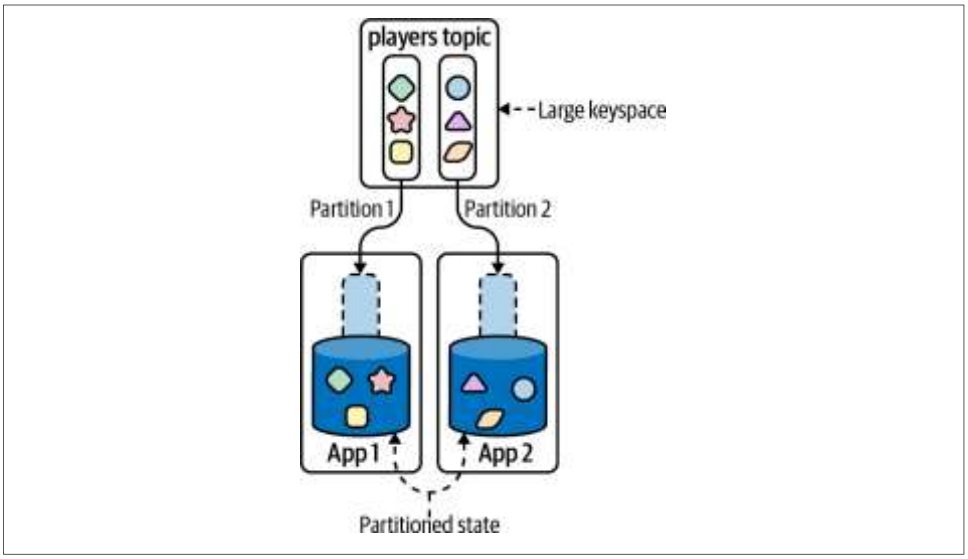

Figure 4-2. A KTable should be used when you want to partition state across multiple application instances and need time synchronized processing

Our updated abstraction table now looks like this:

Kafka topic	Abstraction
score-events	KStream
players	KTable
products	???

We have one topic left: the `products` topic. This topic is relatively small, so we should be able to replicate the state in full across all of our application instances. Let's take a look at the abstraction that allows us to do this: `GlobalKTable`.

GlobalKTable

The `products` topic is similar to the `players` topic in terms of configuration (it's compacted) and its bounded keyspace (we maintain the latest record for each unique product ID, and there are only a fixed number of products that we track). However, the `products` topic has much lower cardinality (i.e., fewer unique keys) than the `players` topic, and even if our leaderboard tracked high scores for several hundred games, this still translates to a state space small enough to fit entirely in-memory.

In addition to being smaller, the data in the `products` topic is also relatively static. Video games take a long time to build, so we don't expect a lot of updates to our `products` topic.

These two characteristics (small and static data) are what `GlobalKTables` were

designed for. Therefore, we will use a `GlobalKTable` for our `products` topic. As a result, each of our Kafka Streams instances will store a full copy of the product information, which, as we'll see later, makes performing joins much easier.

Figure 4-3 shows how each Kafka Streams instance maintains a full copy of the `products` table.

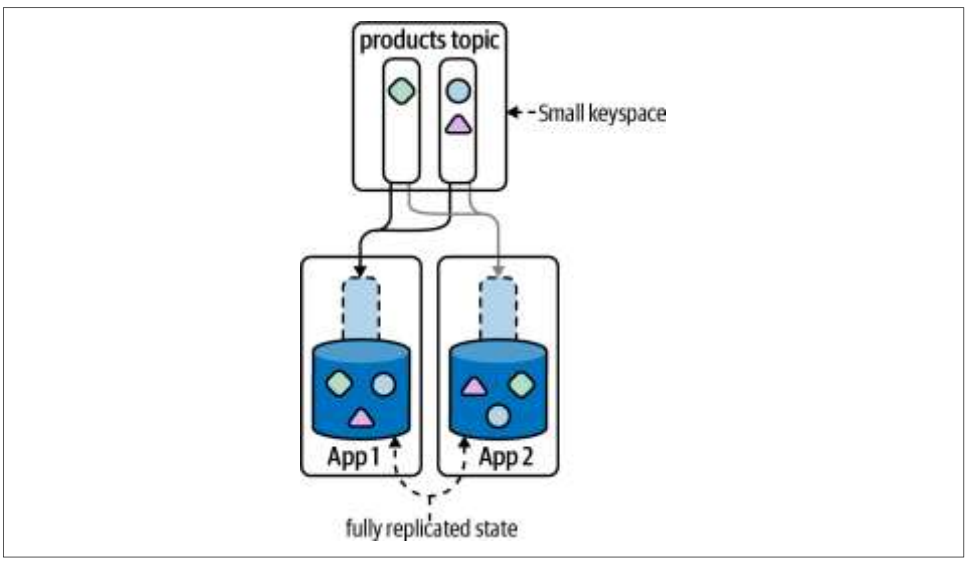

Figure 4-3. A GlobalKTable should be used when your keyspace is small, you want to avoid the co-partitioning requirements of a join, and when time synchronization is not needed

We can now make the final update to our topic-abstraction mapping:

Kafka topic	Abstraction
`score-events`	`KStream`
`players`	`KTable`
`products`	`GlobalKTable`

Now that we've decided which abstraction to use for each of our source topics, we can register the streams and tables.

Registering Streams and Tables

Registering streams and tables is simple. The following code block shows how to use the high-level DSL to create a `KStream`, `KTable`, and `GlobalKTable` using the appropriate builder methods:

```
StreamsBuilder builder = new StreamsBuilder();
```

```
KStream<byte[], ScoreEvent> scoreEvents =
    builder.stream(
        "score-events",
        Consumed.with(Serdes.ByteArray(), JsonSerdes.ScoreEvent())); ❶

KTable<String, Player> players =
    builder.table(
        "players",
        Consumed.with(Serdes.String(), JsonSerdes.Player())); ❷

GlobalKTable<String, Product> products =
    builder.globalTable(
        "products",
        Consumed.with(Serdes.String(), JsonSerdes.Product())); ❸
```

❶ Use a `KStream` to represent data in the `score-events` topic, which is currently unkeyed.

❷ Create a partitioned (or sharded) table for the `players` topic, using the `KTable` abstraction.

❸ Create a `GlobalKTable` for the `products` topic, which will be replicated in full to each application instance.

By registering the source topics, we have now implemented the first step of our leaderboard topology (see Figure 4-1). Let's move on to the next step: joining streams and tables.

Joins

The most common method for combining datasets in the relational world is through joins. In relational systems, data is often highly dimensional and scattered across many different tables. It is common to see these same patterns in Kafka as well, eitherbecause events are sourced from multiple locations, developers are comfortable or used to relational data models, or because certain Kafka integrations (e.g., the JDBC Kafka Connector, Debezium, Maxwell, etc.) bring both the raw data and the data models of the source systems with them.

Regardless of how data becomes scattered in Kafka, being able to combine data in separate streams and tables based on *relationships* opens the door for more advanced data enrichment opportunities in Kafka Streams. Furthermore, the join method for combining datasets is very different from simply merging streams, as we saw in Figure 3-6. When we use the `merge` operator in Kafka Streams, records on both sidesof the merge are unconditionally combined into a single stream. Simple merge opera-tions are therefore stateless since they do not need additional context about the eventsbeing merged.

Joins, however, can be thought of as a special kind of *conditional merge* that cares about the relationship between events, and where the records are not copied verbatim

into the output stream but rather combined. Furthermore, these relationships must be captured, stored, and referenced at merge time to facilitate joining, which makes joining a stateful operation. Figure 4-4 shows a simplified depiction of how one type of join works (there are several types of joins, as we'll see in Table 4-5).

As with relational systems, Kafka Streams includes support for multiple kinds of joins. So, before we learn how to join our `score-events` stream with our `players` table, let's first familiarize ourselves with the various join operators that are availableto us so that we can select the best option for our particular use case.

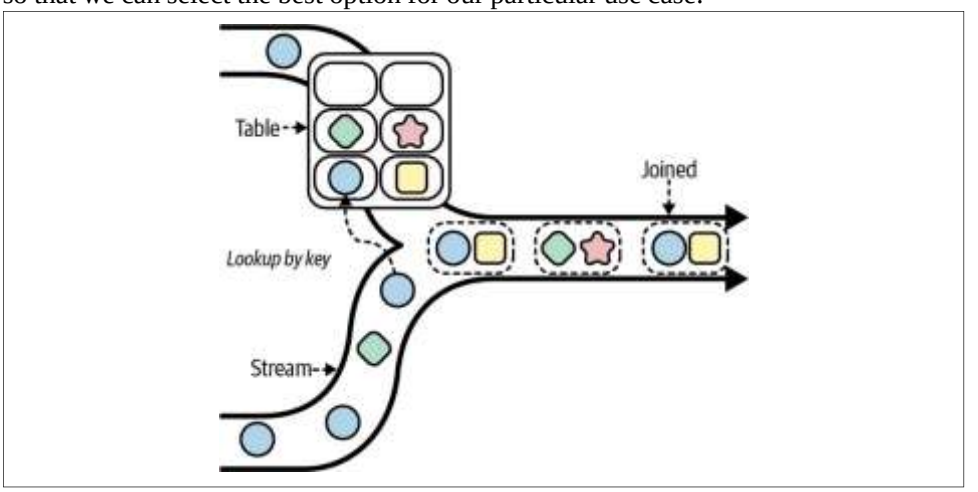

Figure 4-4. Joining messages

Join Operators

Kafka Streams includes three different join operators for joining streams and tables. Each operator is detailed in Table 4-4.

Table 4-4. Join operators

Operator	Description
`join`	Inner join. The join is triggered when the input records on both sides of the join share the same key.
`leftJoin`	Left join. The join semantics are different depending on the type of join: • For stream-table joins: a join is triggered when a record on the *left side* of the join is received. If there is no record with the same key on the right side of the join, then the right value is set to null. • For stream-stream and table-table joins: same semantics as a stream-stream left join, except an input on the right side of the join can also trigger a lookup. If the right side triggers the join and there is no matching key on the left side, then the join will not produce a result.
`outerJoin`	Outer join. The join is triggered when a record on *either side* of the join is received. If there is no matching record with the same key on the opposite side of the join, then the corresponding value is set to null.

When discussing the differences between join operators, we refer to different *sides of the join*. Just remember, the *right side* of the join is always passed as a parameter to the relevant join operator. For example:

```
KStream<String, ScoreEvent> scoreEvents = ...;
KTable<String, Player> players = ...;

scoreEvents.join(players, ...); ❶
```

❶ `scoreEvents` is the *left side* of the join. `players` is the *right side* of the join.

Now, let's look at the type of joins we can create with these operators.

Join Types

Kafka Streams supports many different types of joins, as shown in Table 4-5. For now, it's sufficient to understand that co-partitioning is simply an extraset of requirements that are needed to actually perform the join.

Table 4-5. Join types

Type	Windowed	Operators	Co-partitioning required
`KStream-KStream`	Yes[a]	• `join` • `leftJoin` • `outerJoin`	Yes
`KTable-KTable`	No	• `join` • `leftJoin` • `outerJoin`	Yes
`KStream-KTable`	No	• `join` • `leftJoin`	Yes
`KStream-GlobalKTable`	No	• `join` • `leftJoin`	No

[a] One key thing to note is that `KStream-KStream` joins are windowed. We will discuss this in detail in the next chapter.

The two types of joins we need to perform in this chapter are:

- `KStream-KTable` to join the `score-events` `KStream` and the `players` `KTable`
- `KStream-GlobalKTable` to join the output of the previous join with the `products` `GlobalKTable`

We will use an inner join, using the `join` operator for each of the joins since we only want the join to be triggered when there's a match on both sides. However, before we do that, we can see that the first join we'll be creating (`KStream-KTable`) shows that co-partitioning is required. Let's take a look at what that means before we write any more code.

Co-Partitioning

> If a tree falls in a forest and no one is around to hear it, does it make a sound?
>
> —Aphorism

This famous thought experiment raises a question about what role an observer has in the occurrence of an event (in this case, sound being made in a forest). Similarly, in Kafka Streams, we must always be aware of the effect an observer has on the *processing of an event*.

We learned that each partition is assignedto a single Kafka Streams task, and these tasks will act as the observers in our analogysince they are responsible for actually consuming and processing events. Because there's no guarantee that events on different partitions will be handled by the same Kafka Streams task, we have a potential observability problem.

Figure 4-5 shows the basic observability issue. If related events are processed by different tasks, then we cannot accurately determine the relationship between events because we have two separate observers. Since the whole purpose of joining data is to combine related events, an observability problem will make our joins fail when they should otherwise succeed.

In order to understand the relationship between events (through joins) or compute aggregations on a sequence of events, we need to ensure that related events are routed to the same partition, and so are handled by the same task.

To ensure related events are routed to the same partition, we must ensure the following *co-partitioning requirements* are met:

- Records on both sides must be keyed by the same field, and must be partitioned on that key using the same partitioning strategy.
- The input topics on both sides of the join must contain the same number of partitions. (This is the one requirement that is checked at startup. If this requirement is not met, then a `TopologyBuilderException` will be thrown.)

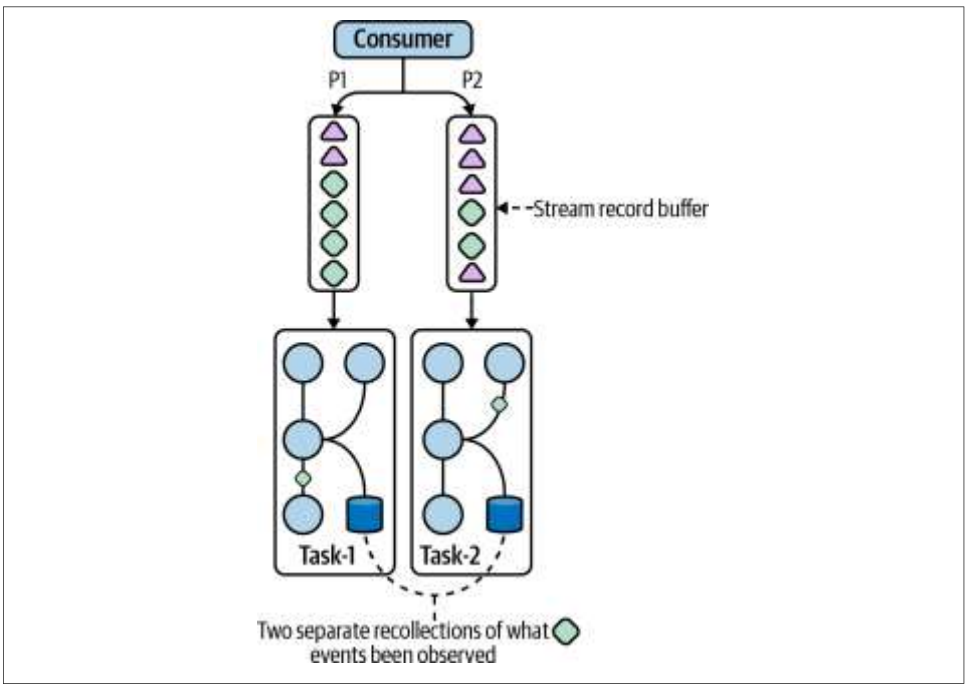

Figure 4-5. If we want to join related records but these records aren't always processed by the same task, then we have an observability problem

In this tutorial, we meet all of the requirements to perform a `KTable-KTable` join except the first one. Recall that records in the `score-events` topic are unkeyed, but we'll be joining with the `players KTable`, which is keyed by player ID. Therefore, we need to rekey the data in `score-events` by player ID as well, *prior to performing thejoin*. This can be accomplished using the `selectKey` operator, as shown in Example 4-1.

Example 4-1. The `selectKey` operator allows us to rekey records; this is often needed to meet the co-partitioning requirements for performing certain types of joins

```
KStream<String, ScoreEvent> scoreEvents =
  builder
    .stream(
      "score-events",
      Consumed.with(Serdes.ByteArray(), JsonSerdes.ScoreEvent()))
    .selectKey((k, v) -> v.getPlayerId().toString()); ❶
```

❶ `selectKey` is used to rekey records. In this case, it helps us meet the first co-partitioning requirement of ensuring records on both sides of the join (the `score-events` data and `players` data) are keyed by the same field.

A visualization of how records are rekeyed is shown in Figure 4-6.

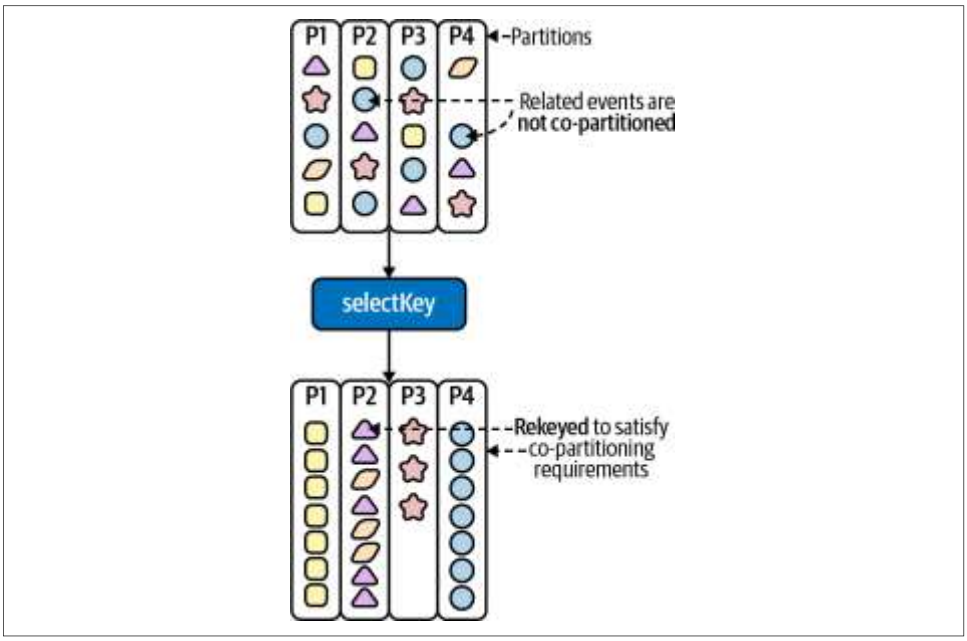

Figure 4-6. Rekeying messages ensures related records appear on the same partition

When we add a key-changing operator to our topology, the underlying data will be *marked for repartitioning*. This means that as soon as we add a downstream operator that reads the new key, Kafka Streams will:

- Send the rekeyed data to an internal repartition topic
- Reread the newly rekeyed data back into Kafka Streams

This process ensures related records (i.e., records that share the same key) will be processed by the same task in subsequent topol- ogy steps. However, the network trip required for rerouting data toa special repartition topic means that rekey operations can be expensive.

What about our `KStream-GlobalKTable` for joining the `products` topic? As shown in Table 4-5, co-partitioning is not required for `GlobalKTable` joins since the state is fully replicated across each instance of our Kafka Streams app. Therefore, we will never encounter this kind of observability problem with a `GlobalKTable` join.

We're almost ready to join our streams and tables. But first, let's take a look at how records are actually combined during a join operation.

Value Joiners

When performing a join using traditional SQL, we simply need to use the join operator in conjunction with a `SELECT` clause to specify the shape (or *projection*) of the combined join record. For example:

```
SELECT a.customer_name, b.purchase_amount ❶
FROM customers a
LEFT JOIN purchases b
ON a.customer_id = b.customer_id
```

❶ The projection of the combined join record includes two columns.

However, in Kafka Streams, we need to use a `ValueJoiner` to specify how different records should be combined. A `ValueJoiner` simply takes each record that is involved in the join, and produces a new, combined record. Looking at the first join, in which we need to join the `score-events KStream` with the `players KTable`, the behavior of the value joiner could be expressed using the following pseudocode:

```
(scoreEvent, player) -> combine(scoreEvent, player);
```

But we can do much better than that. It's more typical to have a dedicated data class that does one of the following:

- Wraps each of the values involved in the join
- Extracts the relevant fields from each side of the join, and saves the extracted values in class properties

We will explore both approaches next. First, let's start with a simple wrapper class for the `score-events -> players` join. Example 4-2 shows a simple implementation of a data class that wraps the record on each side of the join.

Example 4-2. The data class that we'll use to construct the joined `score-events -> players` record

```
public class ScoreWithPlayer {
  private ScoreEvent scoreEvent;
  private Player player;

  public ScoreWithPlayer(ScoreEvent scoreEvent, Player player) {❶
    this.scoreEvent = scoreEvent; ❷
    this.player = player;
  }

  // accessors omitted from brevity
}
```

❶ The constructor contains a parameter for each side of the join. The left side of the join contains `ScoreEvent`, and the right side contains a `Player`.

❷ We simply save a reference to each record involved in the join inside of our wrapper class.

We can use our new wrapper class as the return type in our `ValueJoiner`. Example 4-3 shows an example implementation of a `ValueJoiner` that combines a `ScoreEvent` (from the `score-events KStream`) and a `Player` (from the `players KTable`) into a `ScoreWithPlayer` instance.

Example 4-3. The ValueJoiner for combining score-events and players

```
ValueJoiner<ScoreEvent, Player, ScoreWithPlayer> scorePlayerJoiner =
        (score, player) -> new ScoreWithPlayer(score, player); ❶
```

❶ We could also simply use a static method reference here, such as `ScoreWithPlayer::new`.

Let's move on to the second join. This join needs to combine a `ScoreWithPlayer` (from the output of the first join) with a `Product` (from the `products GlobalKTable`). We could reuse the wrapper pattern again, but we could also simply extract the properties we need from each side of the join, and discard the rest.

The following code block shows an implementation of a data class that follows the second pattern. We simply extract the values we want and save them to the appropriate class properties:

```
public class Enriched {
  private Long playerId;
  private Long productId;
  private String playerName;
  private String gameName;
  private Double score;

  public Enriched(ScoreWithPlayer scoreEventWithPlayer, Product product) {
    this.playerId = scoreEventWithPlayer.getPlayer().getId();
    this.productId = product.getId();
    this.playerName = scoreEventWithPlayer.getPlayer().getName();
    this.gameName = product.getName();
    this.score = scoreEventWithPlayer.getScoreEvent().getScore();
  }

  // accessors omitted from brevity
}
```

With this new data class in place, we can build our `ValueJoiner` for the `KStream-GlobalKTable` join using the code shown in Example 4-4.

Example 4-4. A ValueJoiner, expressed as a lambda, that we will use for the join

```
ValueJoiner<ScoreWithPlayer, Product, Enriched> productJoiner =
    (scoreWithPlayer, product) -> new Enriched(scoreWithPlayer, product);
```

Now that we've told Kafka Streams *how* to combine our join records, we can actually perform the joins.

KStream to KTable Join (players Join)

It's time to join our `score-events KStream` with our `players KTable`. Since we only want to trigger the join when the `ScoreEvent` record can be matched to a `Player` record (using the record key), we'll perform an inner join using the `join` operator, asshown here:

```
Joined<String, ScoreEvent, Player> playerJoinParams =
  Joined.with( ❶
    Serdes.String(),
    JsonSerdes.ScoreEvent(),
    JsonSerdes.Player()
  );

KStream<String, ScoreWithPlayer> withPlayers =
  scoreEvents.join( ❷
    players,
    scorePlayerJoiner, ❸
    playerJoinParams
  );
```

❶ The join parameters define how the keys and values for the join records should be serialized.

❷ The `join` operator performs an inner join.

❸ This is the `ValueJoiner` we created in Example 4-3. A new `ScoreWithPlayer` value is created from the two join records. Check out the `ScoreWithPlayer` data class in Example 4-2 to see how the left and right side of the join values are passed to the constructor.

It's that simple. Furthermore, if you were to run the code at this point and then list all of the topics that are available in your Kafka cluster, you would see that Kafka Streams created two new internal topics for us.

These topics are:

- A repartition topic to handle the rekey operation that we performed in Example 4-1.
- A changelog topic for backing the state store, which is used by the join operator.

You can verify with the `kafka-topics` console script:

```
$ kafka-topics --bootstrap-server kafka:9092 --list

players
products
score-events
dev-KSTREAM-KEY-SELECT-0000000001-repartition ❶
dev-players-STATE-STORE-0000000002-changelog ❷
```

❶ An internal repartition topic that was created by Kafka Streams. It is prefixed with the application ID of our Kafka Streams application (`dev`).

❷ An internal changelog topic that was created by Kafka Streams. As with the repartition topic, this changelog topic is also prefixed with the application ID of our Kafka Streams application.

OK, we're ready to move on to the second join.

KStream to GlobalKTable Join (products Join)

As we discussed in the co-partitioning requirements, records on either side of a `GlobalKTable` join do not need to share the same key. Since the local task has a full copy of the table, we can actually perform a join using some attribute of the record value itself on the stream side of the join,[14] which is more efficient than having to rekey records through a repartition topic just to ensure related records are handled by the same task.

To perform a `KStream-GlobalKTable` join, we need to create something called a `KeyValueMapper`, whose purpose is to specify how to map a `KStream` record to a `GlobalKTable` record.For this tutorial, we can simply extract the product ID from the `ScoreWithPlayer` value to map these records to a `Product`, as shown here:

```
KeyValueMapper<String, ScoreWithPlayer, String> keyMapper =
  (leftKey, scoreWithPlayer) -> {
    return String.valueOf(scoreWithPlayer.getScoreEvent().getProductId());
  };
```

With our `KeyValueMapper` in place, and also the `ValueJoiner` that we created in Example 4-4, we can now perform the join:

```
KStream<String, Enriched> withProducts =
  withPlayers.join(products, keyMapper, productJoiner);
```

This completes the second and third steps of our leaderboard topology (see Figure 4-1). The next thing we need to tackle is grouping the enriched records so thatwe can perform an aggregation.

Grouping Records

Before you perform any stream or table aggregations in Kafka Streams, you must first group the `KStream` or `KTable` that you plan to aggregate. The purpose of grouping is the same as rekeying records prior to joining: to ensure the related records are processed by the same observer, or Kafka Streams task.

There are some slight differences between grouping streams and tables, so we will take a look at each.

Grouping Streams

There are two operators that can be used for grouping a `KStream`:

- `groupBy`
- `groupByKey`

Using `groupBy` is similar to the process of rekeying a stream using `selectKey`, since this operator is a key-changing operator and causes Kafka Streams to mark the stream for repartitioning. If a downstream operator is added that reads the new key, Kafka Streams will automatically create a repartition topic and route the data back to Kafka to complete the rekeying process.

Example 4-5 shows how to use the `groupBy` operator to group a `KStream`.

Example 4-5. Use the groupBy operator to rekey and group a KStream at the same time

```
KGroupedStream<String, Enriched> grouped =
  withProducts.groupBy(
      (key, value) -> value.getProductId().toString(), ❶
      Grouped.with(Serdes.String(), JsonSerdes.Enriched())); ❷
```

❶ We can use a lambda to select the new key, since the `groupBy` operator expects a `KeyValueMapper`, which happens to be a functional interface.

❷ `Grouped` allows us to pass in some additional options for grouping, including the key and value Serdes to use when serializing the records.

However, if your records don't need to be rekeyed, then it is preferable to use the `groupByKey` operator instead. `groupByKey` will *not* mark the stream for repartitioning, and will therefore be more performant since it avoids the additional network calls associated with sending data back to Kafka for repartitioning. The `groupByKey` implementation is shown here:

```
KGroupedStream<String, Enriched> grouped =
    withProducts.groupByKey(
      Grouped.with(Serdes.String(),
      JsonSerdes.Enriched()));
```

Since we want to calculate the high scores for each *product ID*, and since our enriched stream is currently keyed by *player ID*, we will use the `groupBy` variation shown in Example 4-5 in the leaderboard topology.

Regardless of which operator you use for grouping a stream, Kafka Streams will return a new type that we haven't discussed before: `KGroupedStream`. `KGroupedStream`is just an intermediate representation of a stream that allows us to perform aggrega- tions. We will look at aggregations shortly, but first, let's take a look at how to group`KTables`.

Grouping Tables

Unlike grouping streams, there is only one operator available for grouping tables: `groupBy`. Furthermore, instead of returning a `KGroupedStream`, invoking `groupBy` on a `KTable` returns a different intermediate representation: `KGroupedTable`. Otherwise, the process of grouping `KTables` is identical to grouping a `KStream`. For example, if we wanted to group the `players KTable` so that we could later perform some aggregation (e.g., count the number of players), then we could use the following code:

```
KGroupedTable<String, Player> groupedPlayers =
    players.groupBy(
        (key, value) -> KeyValue.pair(key, value),
        Grouped.with(Serdes.String(), JsonSerdes.Player()));
```

The preceding code block isn't needed for this tutorial since we don't need to group the `players` table, but we are showing it here to demonstrate the concept. We now know how to group streams and tables, and have completed step 4 of our processor topology (see Figure 4-1). Next, we'll learn how to perform aggregations in Kafka Streams.

Aggregations

One of the final steps required for our leaderboard topology is to calculate the high scores for each game. Kafka Streams gives us a set of operators that makes performing these kinds of aggregations very easy:

- `aggregate`
- `reduce`
- `count`

At a high level, aggregations are just a way of combining multiple input values into a single output value. We tend to think of aggregations as mathematical operations, but they don't have to be. While `count` is a mathematical operation that computes the number of events per key, both the `aggregate` and `reduce` operators are more generic, and can combine values using any combinational logic you specify.

`reduce` is very similar to `aggregate`. The difference lies in the

return type. The `reduce` operator requires the output of an aggregation to be of the same type as the input, while the `aggregate` operator can specify a different type for the output record.

Furthermore, aggregations can be applied to both streams and tables. The semantics are a little different across each, since streams are immutable while tables are mutable. This translates into slightly different versions of the `aggregate` and `reduce` operators, with the streams version accepting two parameters: an *initializer* and an *adder*, and the table version accepting three parameters: an *initializer*, *adder*, and *subtractor*.

Let's take a look at how to aggregate streams by creating our high scores aggregation.

Aggregating Streams

In this section, we'll learn how to apply aggregations to record streams, which involves creating a function for initializing a new aggregate value (called an *initializer*) and a function for performing subsequent aggregations as new records come in for a given key (called an *adder* function). First, we'll learn about initializers.

Initializer

When a new key is seen by our Kafka Streams topology, we need some way of initializing the aggregation. The interface that helps us with this is `Initializer`, and like many of the classes in the Kafka Streams API, `Initializer` is a functional interface (i.e., contains a single method), and therefore can be defined as a lambda.

For example, if you were to look at the internals of the `count` aggregation, you'd see an initializer that sets the initial value of the aggregation to `0`:

```
Initializer<Long> countInitializer = () -> 0L; ❶
```

❶ The initializer is defined as a lambda, since the `Initializer` interface is a functional interface.

For more complex aggregations, you can provide your own custom initializer instead. For example, to implement a video game leaderboard, we need some way to compute the top three high scores for a given game. To do this, we can create a separate class that will include the logic for tracking the top three scores, and provide a new instance of this class whenever an aggregation needs to be initialized.

In this tutorial, we will create a custom class called `HighScores` to act as our aggregation class. This class will need some underlying data structure to hold the top three scores for a given video game. One approach is to use a `TreeSet`, which is an orderedset included in the Java standard library, and is therefore pretty convenient for holding high scores (which are inherently ordered).

An initial implementation of our data class that we'll use for the high scores aggregation is shown here:

```
public class HighScores {

  private final TreeSet<Enriched> highScores = new TreeSet<>();

}
```

Now we need to tell Kafka Streams how to initialize our new data class. Initializing a class is simple; we just need to instantiate it:

```
Initializer<HighScores> highScoresInitializer = HighScores::new;
```

Once we have an initializer for our aggregation, we need to implement the logic for actually performing the aggregation (in this case, keeping track of the top three high scores for each video game).

Adder

The next thing we need to do in order to build a stream aggregator is to define the logic for combining two aggregates. This is accomplished using the `Aggregator` inter-face, which, like `Initializer`, is a functional interface that can be implemented usinga lambda. The implementing function needs to accept three parameters:

- The record key
- The record value
- The current aggregate value

We can create our high scores aggregator with the following code:

```
Aggregator<String, Enriched, HighScores> highScoresAdder =
        (key, value, aggregate) -> aggregate.add(value);
```

Note that `aggregate` is a `HighScores` instance, and since our aggregator invokes the `HighScores.add` method, we simply need to implement that in our `HighScores` class. As you can see in the following code block, the code is extremely simple, with the `add` method simply appending a new high score to the internal `TreeSet`, and then removing the lowest score if we have more than three high scores:

```
public class HighScores {
  private final TreeSet<Enriched> highScores = new TreeSet<>();

  public HighScores add(final Enriched enriched) {
    highScores.add(enriched); ❶

    if (highScores.size() > 3) { ❷
      highScores.remove(highScores.last());
    }

    return this;
  }
```

```
    }
```

❶ Whenever our adder method (`HighScores.add`) is called by Kafka Streams, we simply add the new record to the underlying `TreeSet`, which will sort each entry automatically.

❷ If we have more than three high scores in our `TreeSet`, remove the lowest score.

In order for the `TreeSet` to know how to sort `Enriched` objects (and therefore, be able to identify the `Enriched` record with the lowest score to remove when our `high Scores` aggregate exceeds three values), we will implement the `Comparable` interface,as shown in Example 4-6.

Example 4-6. The updated Enriched class, which implements the Comparable interface

```
public class Enriched implements Comparable<Enriched> { ❶

  @Override
  public int compareTo(Enriched o) { ❷
    return Double.compare(o.score, score);
  }

  // omitted for brevity
}
```

❶ We will update our `Enriched` class so that it implements `Comparable`, since determining the top three high scores will involve comparing one `Enriched` object to another.

❷ Our implementation of the `compareTo` method uses the `score` property as a method of comparing two different `Enriched` objects.

Now that we have both our initializer and adder function, we can perform the aggregation using the code in Example 4-7.

Example 4-7. Use Kafka Streams' aggregate operator to perform our high scores aggregation

```
KTable<String, HighScores> highScores =
    grouped.aggregate(highScoresInitializer, highScoresAdder);
```

Aggregating Tables

The process of aggregating tables is pretty similar to aggregating streams. We need an *initializer* and an *adder* function. However, tables are mutable, and need to be able to update an aggregate value whenever a key is deleted. We also need a third parame- ter, called a *subtractor* function.

Subtractor

While this isn't necessary for the leaderboard example, let's assume we want to count the number of players in our `players KTable`. We could use the `count` operator, but to demonstrate how to build a subtractor function, we'll build our own aggregate function that is essentially equivalent to the `count` operator. A basic implementation of an aggregate that uses a subtractor function (as well as an initializer and adder function, which is required for both `KStream` and `KTable` aggregations), is shown here:

```
KGroupedTable<String, Player> groupedPlayers =
  players.groupBy(
      (key, value) -> KeyValue.pair(key, value),
      Grouped.with(Serdes.String(), JsonSerdes.Player()));

groupedPlayers.aggregate(
    () -> 0L, ❶
    (key, value, aggregate) -> aggregate + 1L, ❷
    (key, value, aggregate) -> aggregate - 1L); ❸
```

❶ The initializer function initializes the aggregate to 0.

❷ The adder function increments the current count when a new key is seen.

❸ The subtractor function decrements the current count when a key is removed.

We have now completed step 5 of our leaderboard topology (Figure 4-1). We've written a decent amount of code, so let's see how the individual code fragments fit together in the next section.

Putting It All Together

Now that we've constructed the individual processing steps on our leaderboard topology, let's put it all together. Example 4-8 shows how the topology steps we've created so far come together.

Example 4-8. The processor topology for our video game leaderboard application

```
// the builder is used to construct the topology
StreamsBuilder builder = new StreamsBuilder();

// register the score events stream
KStream<String, ScoreEvent> scoreEvents = ❶
    builder
        .stream(
          "score-events",
          Consumed.with(Serdes.ByteArray(), JsonSerdes.ScoreEvent()))
        .selectKey((k, v) -> v.getPlayerId().toString()); ❷
```

```
// create the partitioned players table
KTable<String, Player> players = ❸
    builder.table("players", Consumed.with(Serdes.String(), JsonSerdes.Player()));

// create the global product table
GlobalKTable<String, Product> products = ❹
    builder.globalTable(
      "products",
      Consumed.with(Serdes.String(), JsonSerdes.Product()));

// join params for scoreEvents - players join
Joined<String, ScoreEvent, Player> playerJoinParams =
    Joined.with(Serdes.String(), JsonSerdes.ScoreEvent(), JsonSerdes.Player());

// join scoreEvents - players
ValueJoiner<ScoreEvent, Player, ScoreWithPlayer> scorePlayerJoiner =
    (score, player) -> new ScoreWithPlayer(score, player);
KStream<String, ScoreWithPlayer> withPlayers =
    scoreEvents.join(players, scorePlayerJoiner, playerJoinParams); ❺

// map score-with-player records to products
KeyValueMapper<String, ScoreWithPlayer, String> keyMapper =
  (leftKey, scoreWithPlayer) -> {
    return String.valueOf(scoreWithPlayer.getScoreEvent().getProductId());
  };

// join the withPlayers stream to the product global ktable
ValueJoiner<ScoreWithPlayer, Product, Enriched> productJoiner =
  (scoreWithPlayer, product) -> new Enriched(scoreWithPlayer, product);
KStream<String, Enriched> withProducts =
  withPlayers.join(products, keyMapper, productJoiner); ❻

// Group the enriched product stream
KGroupedStream<String, Enriched> grouped =
  withProducts.groupBy( ❼
      (key, value) -> value.getProductId().toString(),
      Grouped.with(Serdes.String(), JsonSerdes.Enriched()));

// The initial value of our aggregation will be a new HighScores instance
Initializer<HighScores> highScoresInitializer = HighScores::new;

// The logic for aggregating high scores is implemented in the HighScores.add method
Aggregator<String, Enriched, HighScores> highScoresAdder =
  (key, value, aggregate) -> aggregate.add(value);

// Perform the aggregation, and materialize the underlying state store for querying
KTable<String, HighScores> highScores =
    grouped.aggregate( ❽
        highScoresInitializer,
        highScoresAdder);
```

❶ Read the `score-events` into a `KStream`.

Rekey the messages to meet the co-partitioning requirements needed for the join.

2 3 Read the `players` topic into a `KTable` since the keyspace is large (allowing us to shard the state across multiple application instances) and since we want time synchronized processing for the `score-events -> players` join.

4 Read the `products` topic as a `GlobalKTable`, since the keyspace is small and we don't need time synchronized processing.

5 Join the `score-events` stream and the `players` table.

6 Join the enriched `score-events` with the `products` table.

7 Group the enriched stream. This is a prerequisite for aggregating.

8 Aggregate the grouped stream. The aggregation logic lives in the `HighScores` class.

Let's add the necessary configuration for our application and start streaming:

```
Properties props = new Properties();
props.put(StreamsConfig.APPLICATION_ID_CONFIG, "dev");
props.put(StreamsConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");

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

At this point, our application is ready to start receiving records and calculating high scores for our leaderboard. However, we still have one final step to tackle in order to expose the leaderboard results to external clients. Let's move on to the final step of our processor topology, and learn how to expose the state of our Kafka Streams application using interactive queries.

Interactive Queries

One of the defining features of Kafka Streams is its ability to expose application state, both locally and to the outside world. The latter makes it easy to build event-driven microservices with extremely low latency. In this tutorial, we can use interactive queries to expose our high score aggregations.

In order to do this, we need to *materialize* the state store. We'll learn how to do this in the next section.

Materialized Stores

We already know that stateful operators like `aggregate`, `count`, `reduce`, etc., leverage state stores to manage internal state. However, if you look closely at our method for aggregating high scores in Example 4-7, you won't see any mention of a state store.

This variant of the `aggregate` method uses an *internal state store* that is only accessed by the processor topology.

If we want to enable read-only access of the underlying state store for ad hoc queries, we can use one of the overloaded methods to force the materialization of the state store locally. *Materialized state stores* differ from internal state stores in that they are explicitly named and are queryable outside of the processor topology. This is where the `Materialized` class comes in handy. Example 4-9 shows how to materialize a persistent key-value store using the `Materialized` class, which will allow us to query the store using interactive queries.

Example 4-9. Materialized state store with minimal configuration

```
KTable<String, HighScores> highScores =
    grouped.aggregate(
        highScoresInitializer,
        highScoresAdder,
        Materialized.<String, HighScores, KeyValueStore<Bytes, byte[]>> ❶
            as("leader-boards") ❷
            .withKeySerde(Serdes.String()) ❸
            .withValueSerde(JsonSerdes.HighScores()));
```

❶ This variation of the `Materialized.as` method includes three generics:

- The key type of the store (in this case, `String`)
- The value type of the store (in this case, `HighScores`)
- The type of state store (in this case, we'll use a simple key-value store, represented by `KeyValueStore<Bytes, byte[]>`)

❷ Provide an explicit name for the store to make it available for querying outside of the processor topology.

❸ We can customize the materialized state store using a variety of parameters, including the key and value Serdes, as well as other options.

Once we've materialized our `leader-boards` state store, we are almost ready to expose this data via ad hoc queries. The first thing we need to do, however, is to retrieve the store from Kafka Streams.

Accessing Read-Only State Stores

When we need to access a state store in read-only mode, we need two pieces of information:

- The name of the state store
- The type of state store

As we saw in Example 4-9, the name of our state store is `leader-boards`. We need to retrieve the appropriate read-only wrapper for our underlying state store using the `QueryableStoreTypes` factory class. There are multiple state stores supported, including:

- `QueryableStoreTypes.keyValueStore()`
- `QueryableStoreTypes.timestampedKeyValueStore()`
- `QueryableStoreTypes.windowStore()`
- `QueryableStoreTypes.timestampedWindowStore()`
- `QueryableStoreTypes.sessionStore()`

In our case, we're using a simple key-value store, so we need the `QueryableStoreType.keyValueStore()` method. With both the state store name and the state store type, we can instantiate an instance of a queryable state store to be used in interactive queries, by using the `KafkaStreams.store()` method, as shown in Example 4-10.

Example 4-10. Instantiate a key-value store that can be used for performing interactive queries

```
ReadOnlyKeyValueStore<String, HighScores> stateStore =
    streams.store(
        StoreQueryParameters.fromNameAndType(
            "leader-boards",
            QueryableStoreTypes.keyValueStore()));
```

When we have our state store instance, we can query it. The next section discusses the different query types available in key-value stores.

Querying Nonwindowed Key-Value Stores

Each state store type supports different kinds of queries. For example, windowed stores (e.g., `ReadOnlyWindowStore`) support key lookups using time ranges, while simple key-value stores (`ReadOnlyKeyValueStore`) support point lookups, range scans, and count queries.

We will discuss windowed state stores in the next chapter, so for now, let's demonstrate the kinds of queries we can make to our `leader-boards` store.

The easiest way to determine which query types are available for your state store type is to check the underlying interface. As we can see from the interface definition in the following snippet, simple key-value stores support several different types of queries:

```
public interface ReadOnlyKeyValueStore<K, V> {

    V get(K key);

    KeyValueIterator<K, V> range(K from, K to);

    KeyValueIterator<K, V> all();

    long approximateNumEntries();
}
```

Let's take a look at each of these query types, starting with the first: point lookups (`get()`).

Point lookups

Perhaps the most common query type, point lookups simply involve querying the state store for an individual key. To perform this type of query, we can use the `get` method to retrieve the value for a given key. For example:

```
HighScores highScores = stateStore.get(key);
```

Note that a point lookup will return either a deserialized instance of the value (in this case, a `HighScores` object, since that is what we're storing in our state store) or `null` if the key is not found.

Range scans

Simple key-value stores also support range scan queries. Range scans return an iterator for an inclusive range of keys. It's very important to close the iterator once you are finished with it to avoid memory leaks.

The following code block shows how to execute a range query, iterate over each result, and close the iterator:

```
KeyValueIterator<String, HighScores> range = stateStore.range(1, 7); ❶

while (range.hasNext()) {
    KeyValue<String, HighScores> next = range.next(); ❷

    String key = next.key;
    HighScores highScores = next.value; ❸

    // do something with high scores object
```

```
}

range.close(); ❹
```

❶ Returns an iterator that can be used for iterating through each key in the selected range.

❷ Get the next element in the iteration.

❸ The `HighScores` value is available in the `next.value` property.

❹ It's very important to close the iterator to avoid memory leaks. Another way of closing is to use a try-with-resources statement when getting the iterator.

All entries

Similar to a range scan, the `all()` query returns an iterator of key-value pairs, and is similar to an unfiltered `SELECT *` query. However, this query type will return an iterator for all of the entries in our state store, instead of those within a specific key range only. As with range queries, it's important to close the iterator once you're finished with it to avoid memory leaks. The following code shows how to execute an `all()` query. Iterating through the results and closing the iterator is the same as the range scan query, so we have omitted that logic for brevity:

```
KeyValueIterator<String, HighScores> range = stateStore.all();
```

Number of entries

Finally, the last query type is similar to a `COUNT(*)` query, and returns the approximate number of entries in the underlying state store.

When using RocksDB persistent stores, the returned value is approximate since calculating a precise count can be expensive and, when it comes to RocksDB-backed stores, challenging as well.Taken from the RocksDB FAQ:

> Obtaining an accurate number of keys [in] LSM databases like RocksDB is a challenging problem as they have duplicate keys and deletion entries (i.e., tombstones) that will require a full compaction in order to get an accurate number of keys. In addition, if the RocksDB database contains merge operators, it will also make the estimated number of keys less accurate.

On the other hand, if using an in-memory store, the count will be exact.

To execute this type of query against a simple key-value store, we could run the following code:

```
long approxNumEntries = stateStore.approximateNumEntries();
```

Now that we know how to query simple key-value stores, let's see where we can actually execute these queries from.

Local Queries

Each instance of a Kafka Streams application can query its own local state. However, it's important to remember that unless you are materializing a `GlobalKTable` or running a single instance of your Kafka Streams app,[17] the local state will only represent a partial view of the entire application state (this is the nature of a `KTable`, as discussed in "KTable" on page 107).

Luckily for us, Kafka Streams provides some additional methods that make it easy to connect distributed state stores, and to execute *remote queries*, which allow us to query the entire state of our application. We'll learn about remote queries next.

Remote Queries

In order to query the full state of our application, we need to:

- Discover which instances contain the various fragments of our application state
- Add a remote procedure call (RPC) or REST *service* to expose the local state to other running application instances[18]
- Add an RPC or REST *client* for querying remote state stores from a running application instance

Regarding the last two points, you have a lot of flexibility in choosing which server and client components you want to use for inter-instance communication. In this tutorial, we'll use Javalin to implement a REST service due to its simple API. We will also use OkHttp, developed by Square, for our REST client for its ease of use. Let's add these dependencies to our application by updating our *build.gradle* file with the following:

```
dependencies {

  // required for interactive queries (server)
  implementation 'io.javalin:javalin:3.12.0'

  // required for interactive queries (client)
  implementation 'com.squareup.okhttp3:okhttp:4.9.0'

  // other dependencies
}
```

Now let's tackle the issue of instance discovery. We need some way of broadcasting which instances are running at any given point in time and where they are running. The latter can be accomplished using the `APPLICATION_SERVER_CONFIG` parameter to specify a host and port pair in Kafka Streams, as shown here:

```
Properties props = new Properties();

props.put(StreamsConfig.APPLICATION_SERVER_CONFIG, "myapp:8080"); ❶

// other Kafka Streams properties omitted for brevity

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

❶ Configure an endpoint. This will be communicated to other running application instances through Kafka's consumer group protocol. It's important to use an IP and port pair that other instances can use to communicate with your application (i.e., `localhost` would not work since it would resolve to different IPs depending on the instance).

Note that setting the `APPLICATION_SERVER_CONFIG` parameter config doesn't actually tell Kafka Streams to start listening on whatever port you configure. In fact, Kafka Streams does not include a built-in RPC service. However, this host/port information is transmitted to other running instances of your Kafka Streams application and is made available through dedicated API methods, which we will discuss later. But first,let's set up our REST service to start listening on the appropriate port (`8080` in this example).

In terms of code maintainability, it makes sense to define our leaderboard REST service in a dedicated file, separate from the topology definition. The following code block shows a simple implementation of the leaderboard service:

```
class LeaderboardService {
  private final HostInfo hostInfo; ❶
  private final KafkaStreams streams; ❷

  LeaderboardService(HostInfo hostInfo, KafkaStreams streams) {
    this.hostInfo = hostInfo;
    this.streams = streams;
  }

  ReadOnlyKeyValueStore<String, HighScores> getStore() { ❸
    return streams.store(
        StoreQueryParameters.fromNameAndType(
            "leader-boards",
            QueryableStoreTypes.keyValueStore()));
  }

  void start() {
    Javalin app = Javalin.create().start(hostInfo.port()); ❹

    app.get("/leaderboard/:key", this::getKey); ❺
```

```
    }
}
```

❶ `HostInfo` is a simple wrapper class in Kafka Streams that contains a hostname and port. We'll see how to instantiate this shortly.

❷ We need to keep track of the local Kafka Streams instance. We will use some API methods on this instance in the next code block.

❸ Add a dedicated method for retrieving the state store that contains the leaderboard aggregations. This follows the same method for retrieving a read-only state store wrapper that we saw in Example 4-10.

❹ Start the Javalin-based web service on the configured port.

❺ Adding endpoints with Javalin is easy. We simply map a URL path to a method, which we will implement shortly. Path parameters, which are specified with a leading colon (e.g., `:key`), allow us to create dynamic endpoints. This is ideal for a point lookup query.

Now, let's implement the `/leaderboard/:key` endpoint, which will show the high scores for a given key (which in this case is a product ID). As we recently learned, we can use a point lookup to retrieve a single value from our state store. A naive implementation is shown in the following:

```
void getKey(Context ctx) {
    String productId = ctx.pathParam("key");
    HighScores highScores = getStore().get(productId); ❶
    ctx.json(highScores.toList()); ❷
}
```

❶ Use a point lookup to retrieve a value from the local state store.

❷ Note: the `toList()` method is available in the source code.

Unfortunately, this isn't sufficient. Consider the example where we have two running instances of our Kafka Streams application. Depending on *which* instance we query and *when* we issue the query (state can move around whenever there is a consumer rebalance), we may not be able to retrieve the requested value. Figure 4-7 shows this conundrum.

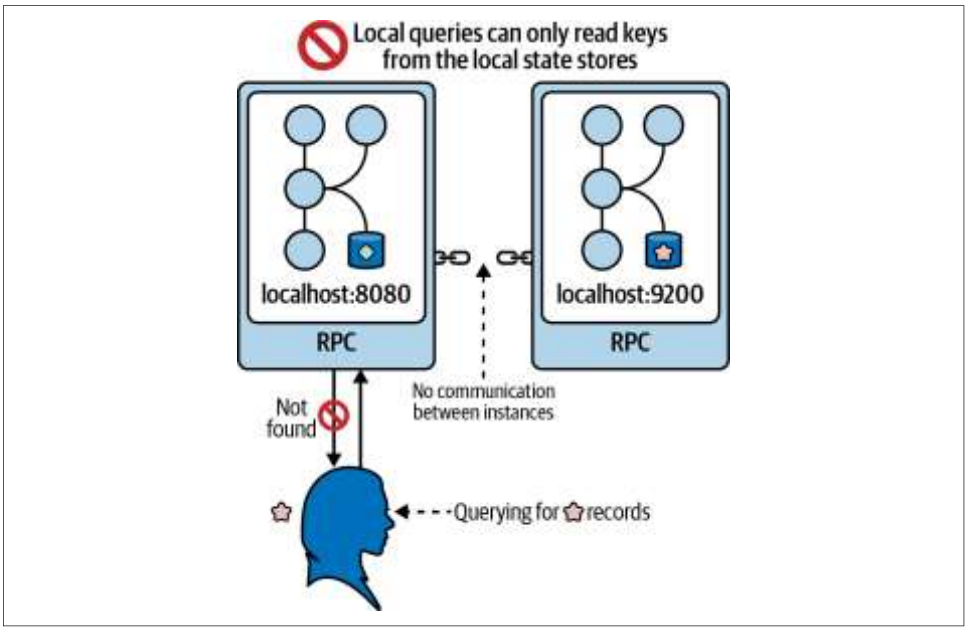

Figure 4-7. When state is partitioned across multiple application instances, local queries are not sufficient

Fortunately, Kafka Streams provides a method called `queryMetadataForKey`, which allows us to discover the application instance (local or remote) that a specific key lives on. An improved implementation of our `getKey` method is shown in Example 4-11.

Example 4-11. An updated implementation of the `getKey` method, which leverages remote queries to pull data from different application instances

```
void getKey(Context ctx) {

  String productId = ctx.pathParam("key");

  KeyQueryMetadata metadata =
      streams.queryMetadataForKey(
        "leader-boards", productId, Serdes.String().serializer()); ❶

  if (hostInfo.equals(metadata.activeHost())) {


    HighScores highScores = getStore().get(productId); ❷

    if (highScores == null) { ❸
      // game was not found
      ctx.status(404);
      return;
    }
```

```
    // game was found, so return the high scores
    ctx.json(highScores.toList()); ❹
    return;
  }

  // a remote instance has the key
  String remoteHost = metadata.activeHost().host();
  int remotePort = metadata.activeHost().port();
  String url =
    String.format(
        "http://%s:%d/leaderboard/%s",
        remoteHost, remotePort, productId); ❺

  OkHttpClient client = new OkHttpClient();
  Request request = new Request.Builder().url(url).build();

  try (Response response = client.newCall(request).execute()) { ❻
    ctx.result(response.body().string());
  } catch (Exception e) {
    ctx.status(500);
  }
}
```

❶ `queryMetadataForKey` allows us to find which host a specific key should live on.

❷ If the local instance has the key, just query the local state store.

❸ The `queryMetadataForKey` method doesn't actually check to see if the key exists. It uses the default stream partitioner[20] to determine where the key *would exist, if it existed*. Therefore, we check for null (which is returned if the key isn't found) and return a `404` response if it doesn't exist.

❹ Return a formatted response containing the high scores.

❺ If we made it this far, then the key exists on a remote host, if it exists at all. Therefore, construct a URL using the metadata, which includes the host and port of the Kafka Streams instance that would contain the specified key.

❻ Invoke the request, and return the result if successful.

To help visualize what is happening here, Figure 4-8 shows how distributed state stores can be connected using a combination of instance discovery and an RPC/RESTservice.

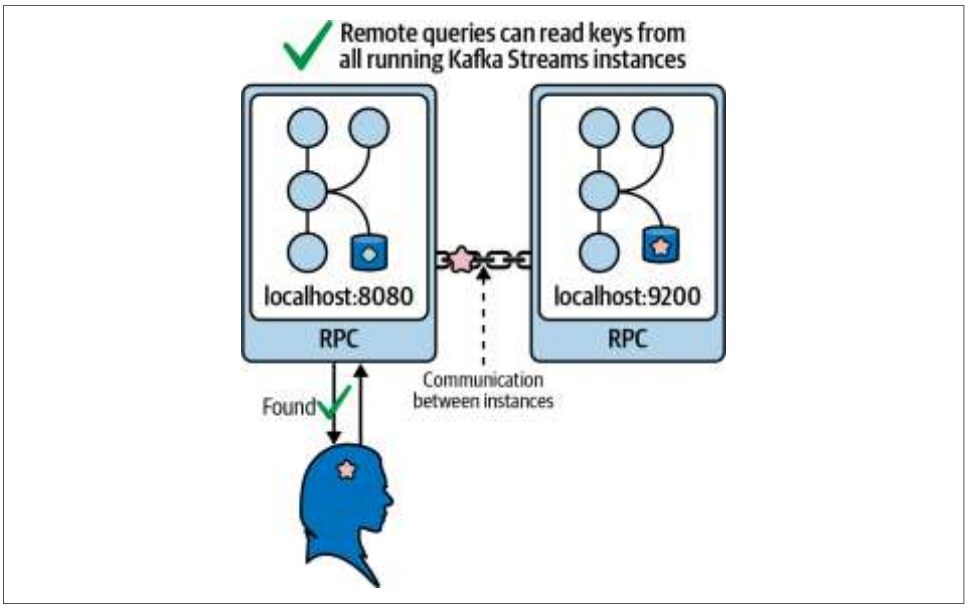

Figure 4-8. Remote queries allow us to query the state of other running application instances

But what if you need to execute a query that doesn't operate on a single key? For example, what if you need to count the number of entries across all of your dis- tributed state stores? The `queryMetadataForKey` wouldn't work well in this case, sinceit requires us to specify a single key. Instead, we would leverage another Kafka Streams method, called `allMetadataForStore`, which returns the endpoint for every running Kafka Streams application that shares the same application ID *and* has at least one active partition for the provided store name.

Let's add a new endpoint to our leaderboard service that surfaces the number of high score records across all of the running application instances:

```
app.get("/leaderboard/count", this::getCount);
```

Now, we'll implement the `getCount` method referenced in the preceding code, which leverages the `allMetadataForStore` method to get the total number of records in each remote state store:

```
void getCount(Context ctx) {
  long count = getStore().approximateNumEntries(); ❶
                                                   ❷
  for (StreamsMetadata metadata : streams.allMetadataForStore("leader-boards")) {
    if (!hostInfo.equals(metadata.hostInfo())) {
      continue; ❸
    }
    count += fetchCountFromRemoteInstance( ❹
      metadata.hostInfo().host(),
      metadata.hostInfo().port());
```

```
    }

    ctx.json(count);
  }
```

❶ Initialize the count with the number of entries in the local state store.

❷ On the following line, we use the `allMetadataForStore` method to retrieve the host/port pairs for each Kafka Streams instance that contains a fragment of the state we want to query.

❸ If the metadata is for the current host, then continue through the loop since we've already pulled the entry count from the local state store.

❹ If the metadata does not pertain to the local instance, then retrieve the count from the remote instance. We've omitted the implementation details of `fetchCountFromRemoteInstance` from this text since it is similar to what we saw in Example 4-11, where we instantiated a REST client and issued a request against a remove application instance. If you're interested in the implementation details, please check the source code for this chapter.

This completes the last step of our leaderboard topology (see Figure 4-1). We can now run our application, generate some dummy data, and query our leaderboard service.

The dummy data for each of the source topics is shown in Example 4-12.

For the keyed topics (`players` and `products`), the record key is formatted as `<key>|<value>`. For the `score-events` topic, the dummy records are simply formatted as `<value>`.

Example 4-12. Dummy records that we will be producing to our source topics

```
# players
1|{"id": 1, "name": "Elyse"}
2|{"id": 2, "name": "Mitch"}
3|{"id": 3, "name": "Isabelle"}
4|{"id": 4, "name": "Sammy"}

# products
1|{"id": 1, "name": "Super Smash Bros"}
6|{"id": 6, "name": "Mario Kart"}

# score-events
{"score": 1000, "product_id": 1, "player_id": 1}
{"score": 2000, "product_id": 1, "player_id": 2}
{"score": 4000, "product_id": 1, "player_id": 3}
{"score": 500, "product_id": 1, "player_id": 4}
{"score": 800, "product_id": 6, "player_id": 1}
{"score": 2500, "product_id": 6, "player_id": 2}
```

```
{"score": 9000.0, "product_id": 6, "player_id": 3}
{"score": 1200.0, "product_id": 6, "player_id": 4}
```

If we produce this dummy data into the appropriate topics and then query our leaderboard service, we will see that our Kafka Streams application not only processed the high scores, but is now exposing the results of our stateful operations. An example response to an interactive query is shown in the following code block:

```
$ curl -s localhost:7000/leaderboard/1 | jq '.'

[
  {
    "playerId": 3,
    "productId": 1,
    "playerName": "Isabelle",
    "gameName": "Super Smash Bros",
    "score": 4000
  },
  {
    "playerId": 2,
    "productId": 1,
    "playerName": "Mitch",
    "gameName": "Super Smash Bros",
    "score": 2000
  },
  {
    "playerId": 1,
    "productId": 1,
    "playerName": "Elyse",
    "gameName": "Super Smash Bros",
    "score": 1000
  }
]
```