Course Overview

Hi, everyone. My name is Eugene Meidinger, and welcome to my course, Building ETL Pipelines from Streaming Data with Kafka and ksqlDB. I'm a business intelligence consultant working at SQLGene Training, LLC. And in this course, we're going to turn Kafka into a stream analytics engine. Some of the major topics that we will cover include streaming data theory, writing ksqlDB queries, and writing Kafka Streams applications. By the end of this course, you'll know how to extract insights from your streaming data living in Apache Kafka. Before beginning this course, you should be familiar with Apache Kafka. I hope you'll join me on this journey to learn how to transform streaming data with the Building ETL Pipelines from Streaming Data with Kafka and ksqlDB course, at Pluralsight.

What Are ksqlDB and Kafka Streams?

What Is Kafka?

Hi, my name is Eugene Meidinger, and I work as a business intelligence consultant. In this course, we'll see how to change Kafka from an event streaming platform to a stream processing platform. We'll be using two tools to do this, Kafka Streams and ksqlDB. We're calling this an ETL pipeline because we'll be transforming the data in a series of linked steps. So let's take a quick overview of things. In this course, we assume that you know about Kafka and have at least played around with it. Even if you haven't, we're going to cover the very basics. Next, we'll talk about ksqlDB, formerly called KSQL. ksqlDB allows you to use an SQL‑style querying language to apply logic and aggregations to a stream of data in Kafka. ksqlDB is easy to use and will cover most of your needs. But you've probably noticed that something is missing here. ksqlDB is a high‑level abstraction, and it's built on top of Kafka Streams, which is a lower‑level API and domain‑specific language for doing stream processing and manipulation. With Kafka, we're going to be focusing in this course on ksqlDB first because it's easier and simpler to get started with. Then we'll talk about Kafka Streams for when you need more advanced capabilities. So ksqlDB and Kafka Streams are technologies. But what about the content? What topics are we going to cover in this course? First, we're going to talk about deployment. ksqlDB allows you to interactively write queries, whereas Kafka Streams is going to require you to write and compile some Java code or potentially some Scala code. Next, we'll be looking at one of the core features of stream processing and especially core to stream analytics, which is aggregations. This is taking many bits of data and consolidating them into summary values. It's extremely common when processing a stream that you won't just be processing events independently of each other. Often you'll be looking at buckets or windows of time. Then we'll look at enriching the data. This is a really fancy term for joining reference or lookup information to add value to the stream of original data. And all throughout the course we'll be covering the challenges of moving from an event streaming platform to a stream processing platform and applying these transformations. So you may be wondering, why bother take this course? Do you really need to spend the next hour or two learning about these tools? And if you just want a storage and distribution platform for your events, then the answer is no. You can go and take a walk in the park or play a video game. But if you want to modify those events, if you want to combine those events, then you'll want to watch this course. Kafka wasn't designed initially for stream processing, and when I say processing, I mean when you make modifications and changes and analyses with those events. Kafka started out as a specialized tool, and only later on were these capabilities added in the form of external libraries and software that interacts with Kafka. If you want to modify, transform, and analyze your data, not just store and transport it, then you need to learn these tools. So what exactly is Kafka? According to the Apache Software Foundation, which maintains Kafka, it is an event streaming platform. So what does that mean, though? What the heck is event streaming? Well, let's walk through this definition, one piece at a time. First, are events. Imagine a newspaper. I know that they're going out of style, but newspapers are a perfect example of what I mean by an event. An event is something that occurred in the past. The past part is key because we track when the thing occurred, just like how a newspaper would have a date on it, and events are purely informational. There's no expectation that will do something in the future. It just sits there as an immutable, unchanging record in the past, if something is wrong in the newspaper, we issue a new correction or retraction, but we never go back and change things. So what about the streaming part? Well, newspapers are huge compared to Kafka events. A Kafka event is usually measured in kilobytes, although the default maximum size is 1 MB. Now for reference, you can almost fit for full‑size tweets into a kilobyte, so this is very, very small. So let's change this from less a newspaper to maybe a news article, and then let's shrink that down. So events are super, super tiny. So now getting back to the streaming part, imagine that instead of a batch collection of articles each day, that we had a constant stream of them. They just keep coming in. This is unbounded. It's an ongoing sequence of events. That is a stream. A stream is an unbounded, continuous sequence of this event data. In Kafka, these streams are called topics. In many ways, this represents more modern new sites, whether we like it or not, in the sense that it's just this firehose of information about things that occurred in the past, and it just keeps coming in. And just like how modern news sites are too fast‑paced for me to keep up with, we often need to make specialized infrastructure such as Kafka to handle the pace and volume of this streaming data. Which brings us to the last part, the platform. All this means is that Kafka is not an end‑user application, but something you build your applications on top of. First, you have Kafka itself. Then you have support for event producers, generating these pieces of information all the time, whether that's every time you like something on social media, or maybe it's an IoT device in a factory. Finally, you have support for event consumers. These are reading this information whenever it's convenient for them. So, you can see now Kafka is an event streaming platform. So that's cool and all, but why would someone choose to use Kafka instead of, say, a traditional relational database? Why add this extra step? There are a number of benefits, but in my mind there are three core reasons to adopt an event streaming platform like Kafka. First is scaling. Because these systems are optimized for one and only one thing, they can not only handle a high volume of events, but also a high velocity of events. In addition to that, systems like Kafka are partitioned, so you can scale out horizontally as well. If the number of events doubles overnight, say for Black Friday, you can double the number of processing nodes if you have enough partitions. Scaling is great and all, but these kinds of systems also represent a shift in design philosophy. And that's the focus on decoupling, loosening the links between different parts of your infrastructure. How can we make it so our systems aren't so interwoven, so that if one of them goes down, we don't take down everything else? This is especially important in the age of containers, where a server may not have a static IP address, a name, or even exist for more than a few hours of work. Kafka allows for decoupling in two ways. First is in space or location. The concept of space, location, and place gets a little bit weird when we talk about the cloud and virtual machines. But imagine for a second a network diagram. That's what I mean by space. By using Kafka as an intermediary, we can swap out data producers and data consumers without the other side knowing about it. This makes migrations and upgrades much, much easier. We can also decouple in terms of time. In a traditional system, if a data source creates the data, it often has to be processed right there and then. But if we add a system in between, we can do a lot of our work when it's cheaper or more convenient. Things like serverless technologies or spot computing allow us to run jobs on demand or when it's cheapest from a cloud perspective. Overall, Kafka adds more flexibility at the potential cost of some increased complexity.

What Is Kafka Streams?

In this section, we're going to talk about Kafka Streams, an API and Java library for stream processing in Kafka. The way that I like to think about Kafka Streams is it's a client library, so goes inside your application, that lets you speak in streams, not in events, just in the same way that you might want to speak with words instead of focusing on every individual letter. I'm currently learning Spanish, and I have to work really, really hard to recognize the words as cohesive units. It's very difficult for me because it's not my native tongue. Kafka Streams allows you to zoom out a little bit and have a broader view of things and talk in a way, so to speak, that's more native to stream processing. So what do we get if we use the Kafka Streams library in one of our applications? First, we get new nouns. With Kafka, our main noun, our main unit of measure, is the event. Remember, these events are tiny, and there's a ton of them. We're talking a few kilobytes, and so this is very, very detailed, very granular. With Kafka Streams, we get to talk about tables and streams, and we can apply our changes and processing to them in all over, as coherent objects. We don't have to think about every single tiny event. We don't have to think about every single grain of sand to work on a beach. We can happily apply a transformation to all of the events in an entire Kafka topic. In addition to nouns, we get verbs. These are the ways that we can manipulate and process the stream. This might be something simple, like filtering out data we don't want, or mapping a very basic function to each event. It might be something more complicated, like breaking up the stream into windows or buckets of time and aggregating that data into events for a new stream. But it's not enough to have now nouns and verbs for our stream manipulation. We want to be able to deploy this into production. We want this to work in production. And so first, Kafka Streams allows you to run multiple copies of your processing application, this is known as application instances, in order to easily scale up your processing power. If it's Black Friday and you need to double the amount of processing you're doing, you can just double the number of application instances. And as long as your data's partitioned correctly, this just works. Kafka Streams handles all that coordination. You don't have to worry about it. Additionally, Kafka Streams provides a way to move work between instances as they scale up and scale down, or if there's a failure. So if one of those application instances goes down for some reason, there's a way for the other ones to pick up the work. So, ultimately, Kafka Streams allows you to work with these streams of data, as well as scale and deploy your processing application in a robust way. So let's imagine that instead of events or news articles, that we have cars on a highway. Kafka is the Department of Transportation, and each road is a Kafka topic. Instead of articles, we have cars, and these cars, they go on and off, and that's all that Kafka cares about. It wants to make sure that there's no potholes, that traffic is flowing smoothly. It just cares about transporting cars. What happens to them doesn't really matter. But now, imagine that each car only had one passenger. Well, fuel wise, this is inefficient. Here in Pittsburgh, we have HOV lanes, or high‑occupancy vehicle lanes, to make sure and encourage that we put more people in the same number of cars so there's not so much congestion and there's better fuel usage. Now imagine, what if we wanted to load a bunch of people onto busses? How would we do this? Well, you can't just have people jump out of a moving car on the highway into the window of a bus and take a seat. This isn't a bad action movie. And so what we have to do is we would have to have the cars exit the highway and go to a bus station and then get on a bus. Then those busses will be able to get back on the highway. So we have the cars, they're going into the bus station, and everybody gets loaded up. And then this bus goes onto the highway again. In this case, again, a highway is just a Kafka topic. That bus station is a data processing application managed by Kafka Streams. It takes data off of Kafka, makes a modification, and then puts it back onto Kafka. This is why it's called a processor. It is processing the events; it is processing the data. Usually this data will go back onto a different topic representing the aggregated data instead of the original raw stream. So what if we wanted to scale up our data processing? Our highway supports multiple lanes, or in Kafka terms, these are partitions. And so when an event is added, it goes to a topic, and it goes onto a specific partition, often based on some sort of key or piece of information. In our case, we can imagine that cars go on a specific lane based on either their color or maybe we want to encourage electric cars, and so we have a lane just for them. And so if we wanted to scale up our bussing operation, we could have two bus stations. Each of them is tied to a specific partition or lane. In Kafka Streams, these parallel bus stations are called application instances. They're copies of the same application, running in parallel and sharing state information as needed to coordinate their work. In addition to scaling out horizontally by increasing our number of processors doing the same work, we can chain, or link, our stream processors. You can imagine an interesting situation where first we get people onto busses, and then we drive them to a dock where we get them onto boats that are even bigger than the busses. This chaining of processors is called a processor topology. It's called a topology because we have nodes, our processors, and directed edges, our streams or topics, that connect then just like in the mathematical subject of a topology and in graph theory. The topology is, what is this shape of all these processors and all these connections between them? In this example, it's very, very simple. We have two nodes and three edges. We had cars, busses and then boats. But you can imagine a much more complex topology or shape, especially when we start joining multiple streams together. This is why the course is titled ETL pipelines, because we have this complex shape of these chained transformations. There's one last concept with Kafka Streams that I want to cover before we move on, and it's something called the table stream duality, which sounds fancy, but it really isn't. A duality is just the mathematical term for two ways of looking at the same thing. Let's imagine that I was playing a board game with my spouse. We could, if we were a little bit odd, track each change in our score as an event. First, we would start with 2 events showing our 0 starting score. Then maybe I get an early lead with 5 points, but my spouse starts to catch up, and even worse, I lose 2 points because of some sort of event card. So now we have our event stream, and the question is, what's our score? Can you tell from looking here what our score is? We have to calculate it from scratch. So we have to go to the first event and just read each and every one. This is time consuming, and you can imagine how long it could take if we have an event stream, if we have a topic that's been going on for days or weeks. Even worse, what if my spouse gets another point? What we haven't finished the game but we have to start all over again, and then we have to reread the whole stream if we haven't been keeping track. So how can we avoid having to reread the entire stream over and over again just to figure out what the current state of play is? For cases where we only care about the most current values, we can take this stream and split it out by key. In this case, the key is player. And we can create a table with a row for each key. Here we have five keys for five potential players. This is called the table stream duality. These are two ways of viewing the same data. If we look at it from the top, then we have a linear sequential stream of events over time, just like you're used to, representing a history of occurrences. But if we look at it from the side, we have a table summarizing the current totals. And in Kafka Streams, you can move back and forth between these two representations. You can take a table in Kafka Streams and turn it back into the change log that got you there.

What Is ksqlDB?

In this section, we'll talk about ksqlDB and how it builds on top of Kafka Streams and Kafka. KsqlDB, formerly called KSQL, is a piece of software created by a company called Confluent that does a lot of work with Kafka. On their site, they define it as an event streaming database built to create stream processing apps on Kafka. That's a lot. That's practically a paragraph, in my mind. So let's shorten this definition a bit. See, this is the key part, ksqlDB, DB stands for database, is a database built on Apache Kafka. That's the important part. We're using Kafka as the base, and we're trying to build something similar to a database on top of it. But what's database? Well, let's compare it to a different product. What is MySQL? What is Postgres? If you've worked in software development, especially with open source software, open source tools, then there's a good chance you've used one of these. And you might say, well, it's It's a relational database that uses Structured Query Language, or SQL for short, which is true. But that doesn't tell us a lot about the insides. Let's start thinking in terms of internals. Well, first, at the very bottom of a database is the storage engine. You need a way to store and save the data in a durable, resilient fashion. Without this, you don't have a database. You have a caching engine, which is fine. Caching tools like Redis are extremely popular. But if you want a true database, then you need a storage engine so that if somebody pulls an electrical cord out and the server goes down, you still have your data. Next, you need a way to access the data. You need an execution engine. This is the part of the database that works with the storage engine and does all the computational work. This is generally the part of the database that is using all of your CPU. Execution engines can be somewhat rigid sometimes. They need very precise instructions, sometimes called an execution plan in some database systems like MySQL. In order to make database systems more accessible for humans who aren't going to give super, super precise commands, we want a query engine. This part will turn database queries, generally in an SQL‑type language, into execution plans or reject them for using invalid syntax. These execution plans that are created are often cached, and in some systems like MySQL, the results of these queries are cached as well. So now we have a database server, but we need a way to talk to it. We need a user interface. Sometimes this is a graphical application such as MySQL Workbench or pgAdmin. Often in open source databases like MySQL or Postgres, there's also a command line user interface which allows for running queries. For MySQL, it's called MySQL, and for Postgres, it's called psql. This is really, really common when you're dealing with open source databases because oftentimes they run on Linux. Oftentimes they run on servers that don't have a desktop available. Finally, we need a language to communicate with the server. Now, you might say, oh, we'll just use a Structured Query Language, or SQL for short. But each database has its own unique flavor of SQL, custom functions, custom features, custom syntax. Only the most simple queries can be copied from one database system to another without issue. They're all based on ANSI SQL standards and are very similar in function. So how does ksqlDB fit into this framework? First, we have our storage engine, and there's two parts to this when we're dealing with ksqlDB. First is Kafka. As I said multiple times, ksqlDB is based on Kafka, and it uses Kafka to durably and reliably store all of the events and main information. For things that are more temporary or transient, it uses RocksDB. And this is important whenever you want to store state information, which is really necessary when we want to do our aggregations or share work between processing nodes. Next, our execution engine is Kafka Streams. This is the library that's doing all of the work. And then finally on top, we've got ksqlDB, which is acting as an intermediary between us and Kafka Streams. In terms of how we work with that interface, we've got two main options. KsqlDB runs a web server, and so this is going to be a common way that you're going to run your queries. You're going to go into the web server, and you're going to execute a query. But there's also a command line interface called KSQL. And so if you have a terminal, you can run it this way. Now, just like something with MySQL Workbench, you're going to be sending the query over the internet to the server. But this is how ksqlDB makes up all these components of a database. It leans heavily on Kafka and Kafka Streams for the storage engine and the execution engine, respectively. And then it just adds a thin little layer to allow us to run SQL queries instead of having to write code. So let's recap here. What is ksqlDB? It's a SQL variant, or SQL variant. This just means that it provides an SQL syntax with some custom functions that make sense in the context of the kind of work that we're doing, such as breaking up the data into windows. It provides a command‑line interface, and if you don't want to use that, there's a web server. So these are the two main ways that you're going to be able to run your queries. Finally, it provides the top layer on this pyramid that's built on proven and tested technologies.

Summary

Let's review what we've covered in the module. At the very foundation, we have Kafka. Kafka is an event streaming platform which stores our events, and events are tiny pieces of data about things that occurred in the past, kind of like news articles. If we want to turn those articles back into a newspaper, if we want to aggregate the data, or maybe if we just want to make modifications, then we're going to want to use a tool like Kafka Streams, which simplifies modifying, transforming, and analyzing the stream as a whole. It simplifies some of the extracting, transforming, and loading in the term ETL. It gives us concepts like streams and tables, instead of having to deal with what's basically a tiny grain of sand known as an event and having to deal with them one by one. Using concepts like application instances and processor topologies, we can scale out and use more complex pipelines without having to worry about the exact details. KsqlDB takes these tools and infrastructure and gives us an SQL language and a friendly user interface. One of the big benefits is we don't have to compile our code, and there's just less work overall. In the rest of this course, we'll look in depth at using ksqlDB and Kafka Streams.

Configuring ksqlDB and CLI

Installing the ksqlDB Server and the CLI

In this module, we're going to talk about getting the infrastructure that you need to work with ksqlDB Thankfully, getting started is extremely easy if you've ever used Docker before. We're going to be touching on three topics in this module. How do we install ksqlDB and the CLI client? How do we make changes to the settings? This can be especially confusing if you're using Docker containers for the first time and you don't have access to a file system or virtual machine like you might be used to. However, we'll see how to get around that and how to configure our Docker containers. These steps will allow us later to write and deploy our KSQL queries. The key takeaway of this module is that ksqlDB is complex software and that not only do you need to install the server and potentially a command line client to talk to it, but that it also depends on a working Kafka setup to be of any use. Usually, I hate doing these kinds of demos because getting the lab set up properly is a giant pain. There's a bunch of different parts you have to get working. But thankfully, ksqlDB is extremely easy to install and configure if you use Docker to do it. This is partly because of how powerful Docker is and partly because the team over at Confluent has done a lot of work providing Docker images and a quick start file for Docker Compose. The ksqlDB software itself has very few requirements for installing it. According to the official FAQ, it has only two. First, you need a Java runtime environment. This makes a lot of sense since it uses Kafka Streams, which is a Java library. Additionally, you need Kafka, which also makes sense, since the whole point of this software is interacting with Kafka, specifically reading from it and writing back to it. And that's it in terms of software dependencies. So how do we actually install it? Well, there are three main ways ordered by increasing difficulty. The absolute easiest way is Docker Compose. Docker Compose is a program designed for building and running applications that depend on multiple Docker containers. If you have Docker set up and installed, this means you just need a compose file, which Confluent has already created for us and one or two commands to get the whole ksqlDB set up, ready for testing The next option is the Confluent platform. Confluent is the company that created ksqlDB, and they provide a software package built on top of Apache Kafka designed to add features and ease the implementation of it. There's a basic version that's free, and it's limited largely by scalability constraints, such as a maximum of 10 partitions and a maximum of 5TB of storage. If you download the Confluent platform, the ksqlDB binaries are included with it. Finally, the source for ksqlDB is openly available on GitHub. My understanding is that you're able to compile ksqlDB from scratch, but I was unable to find any instructions on how to do so. My impression is that they figure if you want to compile it, you probably have enough experience with Java and running Maven builds that you know what to do. So I would not recommend this approach unless that's the case. So let's say you decide to go the Docker Compose route. What containers will the ksqlDB quick start file spin up for you? It depends a lot on whether or not you have any existing infrastructure. If you already have a Kafka server, then you only need two containers. The first is the KSQL command line interface. This is a client piece of software that's optional that allows you to remotely run queries against the server. This is a convenient way to be able to run queries, but it's not 100% necessary because you also have the ksqlDB server. This has a web interface that functions similarly to the command line interface. The ksqlDB server container interprets the KSQL queries and runs the Kafka Streams jobs. Also, depending on how it's configured, it may have that web‑based interface that we talked about, or it could be running in something called headless mode. These two items here are the bare minimum that you need if you have an existing infrastructure. But let's say you're learning, which you probably are, and you don't have Kafka set up. Then you'll need a couple more things. First, you'll need a container running Kafka. Kafka then depends on a container running Zookeeper, which Kafka uses for storing some configuration information and for managing coordination when it's scaling out to multiple nodes. Kafka is a distributed platform, and instead of reinventing the wheel, it uses Zookeeper to handle all the coordination between the distributed parts. You may optionally want Schema Registry. In order to work, ksqlDB needs to be able to deserialize or convert into objects or data structures the data that's in Kafka because initially it's stored as just binary. That's the way that Kafka works. And you want to be able to turn that binary into columns and fields to act upon. And by default, ksqlDB only supports JSON data. If you want to use serialization protocols, like Avro or protobuf, you need Schema Registry to store those schemas, to tell KSQL which bits are which columns. Schema is basically a definition for how the data is being serialized and deserialized in any specific case. If your data has 10 columns, that column information needs to be defined and stored somewhere, in this case Schema Registry.

Demo: Installing ksqlDB and the CLI

So let's take a look at how we might install ksqlDB and the command line interface. Before we jump right into the demo, I want to touch on a couple of quick concepts. First is something called a container. The word container is very generic. I mean, we might as well say box or folder or whatever, but I want you to think of a shipping container used to transport goods across the ocean. Shipping containers are modular and standardized. They're all the same size, and they all are interchangeable. This revolutionized the shipping world by making it easy to fit as many goods as possible onto a ship in a consistent way. In the same way, application containers have made it so you can fit is many applications on the same host machine as possible while still having an isolated environment between them. Containers are built on images. Images are templates for creating containers and are built in layers. They're built on top of each other. So you might have a base Ubuntu image for your Linux operating system and then a Confluent image built on top of that that's been tweaked to work with a variety of pieces of software. So the Confluent image might have Java installed, for example, because Java is important to a lot of these pieces of software. And then finally, you might have a ksqlDB image built on top of that that only contains the changes needed to add that specific piece of software. This allows for standardization by reusing those base or foundational images and only having to change them in one place. Next, we have Docker, which is a container service that manages all of the logistics and details of building and running Docker containers. But there are other container services available if you don't want to use Docker. Finally, we have Docker Compose, which is a tool for running applications that depend on many Docker containers. Docker Compose allows us to store all of our configuration information in one place, as well as starting all the containers in the right order and taking into consideration different dependencies between those containers. So with that out of the way, in this demo, we're going to use Docker containers and Docker Compose to spend up a full development environment for this course. First, we're going to grab the quick start file from Confluent. This references the Docker images that are provided by Confluent and handles basic configuration for the application as a whole. Next, we'll use Docker Compose pull to pull or download the images so that we have the data that we need to build them. And these images are built one layer at a time, which we'll see in the demo. Finally, we'll turn on our application and run the Docker containers by using docker compose up. So let's see what we need to do to get started with ksqlDB. We're going to install it using the quick start. So if you want to pull this up, you can go to ksqldb.io/quickstart.html, or you can just google ksqlDB quick start. So if we scroll down, we can see that we can choose to do a standalone install, or we can depend on one of the platforms that Confluent provides. Here, I want to do a standalone install, and I have some choices depending on whether or not I already have a Kafka broker set up. In this case, I don't have anything set up. So what I would do is I would take this text and I would copy it into a docker‑compose.yml file. And here, I'm using Visual Studio Code because on Windows it has integration with Docker and with PowerShell. So here we can see that what we've got is we've got a file representing multiple Docker containers. I want to show you how to get this up and running. So here we have our broker, and we can see that the image is based on the cp‑enterprise‑kafka image version 5.5.1. Well what I can do is I can see that version 5.5.1, and I can say you know what, let's delete it. And it's saying, oh, are you sure you want to do that? And I'm sure. So now what I can do is in the folder that has my docker‑compose file, I can run docker compose and then pull. Now this happens automatically if you run docker compose up. But what it's going to do is it's going to pull all the layers of my image. And you can see that for four of those layers, those base layers, we already have them because they're shared among all of these images. But for the four specific to Kafka, we have to redownload them. And so it's going to take about 30 or 40 seconds. And the nice thing is you don't have to do this every single time. Once you have the images ready to go, spinning up a container is really, really fast. And we're going to skip ahead through some of this because you really don't need to see a whole download bar. Okay, awesome. And you can see over on the left that we've had our enterprise Kafka appear back as an image. And what I can do is I can tell the shell to clear out. And now what I can run is docker compose up. And again, if I were to run this, it's going to run everything. I wouldn't have to the pull by itself, but I wanted to show you that as a separate steps so it didn't get lost in all the noise. So if I do docker compose up, you'll notice at the top left in the Containers section, we can see that they're starting up. And you'll notice in the terminal, we have all these outputs that are different colors. It's a little hard to read because there's a lot going on. Those different colors and names represent each of these different containers. And so if you're not familiar with Docker containers, they're kind of like little baby servers. These are all running on my machine. They're not taking up all of the OS kind of space and cruft that you would have with normal virtual machines, but they're all separate environments that are connected by a network bridge. And so now everything's up. So what we can do, we'll start a new shell, and I can run docker execute, and I'm passing in the it command because I want it to be interactive. And I'm telling it, hey, I want to run that ksqldb‑cli container, and I want to run this command. The command is ksql and then the name of our server. And so if I go ahead and run that, what it's going to do is it's going to open up a connection in that container for me. And now we can see we've got the command line interface for ksqlDB. So I can run show all topics. Oh, I forgot. All of the commands in ksqlDB have to be ended with a semicolon. So it's pretty strict about this compared to other SQL languages. Great. And now we can see that there's the default processing log and then two hidden logs that you wouldn't normally see, two hidden topics that you wouldn't normally see if you said show all topics. But the key point here is we can see that all of our containers are working. We're able to talk to our Kafka broker using the command line interface. So we have the command line interface, which talks to the ksqlDB server, which talks to the Kafka broker, which depends on Zookeeper for handling any distributed tasks. And it's that easy. It took us a couple minutes to get up and running with ksqlDB.

Configuring the ksqlDB Server and the CLI

It's not enough to just install ksqlDB. You have to specify some configuration settings if you want to use it all. At a bare minimum, you have to tell it where you're Kafka broker is. There are actually quite a few different ways to configure this software. Between the ksqlDB server and the command line interface, you have four ways to configure settings for one or the other. We'll start with more permanent options and move to more ephemeral or temporary options. The first option we have available is to use a configuration file. Under the ksqlDB folder, there should be a dsql‑server.properties file that is read by default. This file uses the syntax of standard Java property files, and it's pretty easy to read. A file‑based approach makes a lot of sense when you have a traditional operating system and server running your application. Files are easy to edit and maintain. They're also persistent, so ideally, once you set them, you don't have to worry about them. However, whenever we move into the world of containers in DevOps, file‑based configuration like this isn't as convenient. Containers are made out of these file system layers built up as we create the image and so we don't want to have to manually edit a file. Thankfully, in that case, we can use something called environment variables. Environment variables are parameters or variables that are provided by the environment the program is being run in. Oftentimes, this might be a shell like bash or PowerShell. The nice thing with these is that they're super easy to set when you're using Docker, and they're not too hard if you're using a regular shell either. If you're using Docker Compose, then they would be specified in the docker‑compose file. If you're manually starting up a Docker container, you can specify them in the docker run command you run by including a ‑e or a ‑‑env parameter. The third option is to use a command line parameter when calling the application when you're starting it up. If you've ever run a command in the command line and added a ‑parameter.command like we just talked about with Docker, then you know what this is. For the ksqlDB Server, you will usually be using a command line parameter when you want to run ksqlDB in headless mode and specify which query to run at startup. The ksql command line interface also supports command line parameters. Finally, when you're actually interacting with the software and you're using a user interface, you may want to set certain per‑session properties. These are very temporary properties of your connection at that point in time. A common example is the auto offset reset, which is a mouthful, which says that when you're running a query against a Kafka topic, should you start at the latest point, which is the default, or should you run the query from the beginning of the topic and read all of the data that's there? As a reminder, these represent a progression from more permanent or fixed options to more targeted or temporary options. And as we go down, these can be useful for overriding the default settings or for running the software in a slightly different mode than usual for a one‑off instance. So let's go through each of these options now. The very first thing that ksqlDB is going to check for when it's looking for settings is the ksql‑server.properties file. This file follows the Java properties format. You'll notice that the default file has comments for headers, as well as comments describing what the specific properties do. Comments start with a hash or pound sign, which is common in a lot of scripting languages. Then, you specify the property name and what value it will take with an equal sign in between. So here we're saying that the bootstrap servers are running on the same machine, also known as localhost, and on port 9092. This makes sense as a default value, but you would never run it like this in production. This is an example of something you might change between a development environment and a production environment. One of the benefits of file‑based configuration is that you can have a different config file for each environment. One of the other things you might notice is that in the default file, some of the settings are commented out. Here I've indicated this with purple. This is done as a convenience by the developers so that you just need to update the setting and uncomment it to make it active. While it has that hash or pound symbol in front of it, ksqlDB ignores it. So what if we want to use containers like with Docker, or what if we want to dynamically inject our configuration, so to speak, instead of depending on files? Well, what we can do is set environmental variables. If you're using a Docker Compose file, each container will have an environment section where we can specify these settings. You can also set them with your shell. This allows us to have reusable Docker images or templates and change just the key settings that we need to on the fly. Docker Compose files used the YAML syntax, which is a plain text format. Here we can see that the settings and values are separated by a colon. There are other ways to set environmental variables like I mentioned, but in this case, it's really convenient to have a file that describes our entire set of containers, including these settings. Now occasionally you might be running the software manually, or you might be using a script to call the software, and normally you would pass the settings file as a parameter to handle all of this configuration. However, there are cases where it makes sense to override a piece of that configuration. For example, you might pass a special parameter to set what queries to run automatically. This kind of behavior is extremely common in command line applications, especially on Linux servers. Finally, if you're using the KSQL command line interface, what if you want to change a setting on the fly? Well, you can use the set command inside of ksqlDB to change the setting. Here we're changing which Kafka server we're bootstrapping off of and then specifying that our query should creed from the beginning of a topic instead of just with the latest data.

Demo: Configuring the ksqlDB Server and the CLI

In this demo, we'll cover three of the ways to set configuration settings. First, we'll use Docker Compose to specify environment variables that ksqlDB will read upon starting. Next, we'll change the command we run by adding command line parameters. Finally, in the CLI, we'll specify some per‑session properties. Now what you see here is Visual Studio Code. It's a text editor that also has a number of extensions available, and I like it a lot when I'm working on Windows because you can see over on the left is the Docker extension. And so we can see that I've got some containers up and running. Now what we're looking at here on the main screen is the docker‑compose file, and this is going to be the ideal way to configure settings if we're using Docker as our deployment method. So what we have here is we have our reference for the ksqlDB Server image. And let's imagine that we want to change the port number that it listens to for clients to connect. So what I can do is I can go down to environment here, and I can change the port number for listeners. And so this is going to be injected into whatever shell or whatever environment is running the ksqlDB Server. I would also want to update the port's configuration. And what this does is it just maps the internal ports of the container to the external ports of the container. That way, if we have something outside the container trying to talk to it, the ports get mapped correctly. And so any client that I might have on my own personal machine could connect to it. So we would save the changes to the file. And then what we would do is we would run docker compose up. To save us some time because you don't want to watch a bunch of text fly by, I've already done that. And we can see I have highlighted there the ksqlDB Server is running on port 8089 You may be thinking, okay, but what are my other options? So if we go back to PowerShell, if I were to run this command here, what I've done is I've run docker exec. So it's going to run a command on a specific container. And what I'm telling it is run the bash command, basically open up a shell for me to be able to type into, a terminal, against the ksqlDB Server, and do it in the etc/ksqldb folder. And so now what I have here is I can expand this up is I have a terminal inside of the container, and I can say okay, what's in this folder? And we've got a lot of different things here, but what I can do is I can use the more command to read ksql‑server.properties, and we can see here's the default configuration file. And so there's that listeners field, and so it's set the default to 8088, and we've overridden it. We've used the environment variables to overwrite what's in this file. And so we can, you know, page through and see some of the other pieces. And so, like we've talked about before, we've got a lot of these configuration settings that are stored in this file. And some are commented out, and some are uncommented. But the problem is that we're dealing with Docker. And so let's say that I wanted to edit one of these files. Well, I might say okay, well, I'll just run nano. I don't have nano. Maybe I'll try to run vi or vim or emacs. I don't have any of that because when I'm dealing with Docker, this image is based on the smallest version of Linux necessary, and so you don't have a lot of these things that you might normally need. In fact, if I were to try and make changes to a file here, it's not going to persist for the next time that I start it up. What I would have to do if I wanted to do it this way is I'd have to make my own custom Docker image that's a layer on top of whatever image I'm working off of. And in a lot of cases, that's not ideal. If you were dealing with a server, this would be great, but that's not what's going on here. So let's get out of this. We'll clear it out. So those are two of the main options when we're dealing with the ksqlDB Server itself. Now what if we're dealing with the command line client? So here I'm going to run a command against the ksqlDB CLI container, and I'm going to tell it run the ksql command. And we get an error because we didn't tell it what listener to point to. It doesn't know. And so the way that we get around that is we can pass it a command line argument. So we could edit the file that's on the container that's telling it to go to localhost 8088. But again, we would have to make our own container like I said. Or what we can do is whenever we're telling it hey, run the ksql command, we can also tell it run this command line argument as well. So I'm going to get out of here, and then I will run this command, and now we're connected. And then finally we can run a per‑session command So what I can do is I can set auto off reset to earliest. And all that means is that we're going to read data from the beginning of a topic instead of the latest point. And we can see that we've successfully set that property. So those are the four different ways that you can configure these programs. You can modify files, which when you're dealing with Docker is a pain because you have to make your own custom images. You can edit the environment settings, the environment variables, which with Docker Compose is just a couple lines. You can use different command line arguments and parameters when you're running the program manually. Or if you're dealing with the command line client, you can specify per‑session settings. It's that easy.

Summary

In this module, we covered a lot to be able to get you up and going with ksqlDB. We covered the different ways to install the software so that you can actually use it. The way that we focused on was Docker and Docker Compose. This is super fast, and depending on your internet speeds, you can spin up a development environment in a few minutes with Docker. Another option, which we didn't cover, would be to download the Confluent platform, which includes the binaries for ksqlDB. This also works, but it takes a little bit more effort. Finally, if you don't mind getting your hands dirty, because the software is open source, you could always compile it from scratch, but I wouldn't recommend that. Now just because you have the software installed, it doesn't mean it knows where your Kafka broker is. So we need to be able to configure the software. Two of the ways involve changes to the operating system. First is to edit the properties file. The other is to change the environment variables. We also looked at ways to configure the software when you're actually running it. First are command line arguments or parameters, and the second are per‑session settings that you would run inside of the application.

Deploying ksqlDB Queries

Comparing ksqlDB to MySQL and Postgres

In this module, we're going to show you how to write your first ksqlDB query and then how to deploy it. First, we'll see that the SQL language for ksqlDB is just plain SQL, plus a few features on top to support stream manipulation. So if you're familiar with MySQL, Postgres, or other relational databases, you should be comfortable using ksqlDB. We'll see what a simple ksqlDB query might look like. Then we'll cover two out of the three ways to run that query. First, we can use the command line for running queries remotely, and we're going to have a demo to show you exactly how to do this. Next, you could use the web server provided by ksqlDB to also run in interactive mode, similar to using the command‑line interface. Finally, we'll cover using something called headless mode, or application mode, where the server runs certain queries at startup, and the users are not interacting with the ksqlDB server directly. When you're trying to learn ksqlDB and the SQL syntax, the key thing to understand is that as a developer ksqlDB queries are basically ANSI standard SQL, plus some missing features relating specifically to working with streams of data. If you've worked with SQL before, then this should be easy to pick up. Now, if you aren't familiar with the history of databases, the idea of a relational database and the SQL language itself were both invented in the '70s. Then, in 1986, it became a standard of the American National Standards Institute, or ANSI, and a year later, the International Organization for Standards, or ISO, also adopted this standard. The first major revision occurred in 1992, and you'll often see SQL‑92 support as a selling point for certain products like database connectors. According to the developers of ksqlDB, it is heavily based on SQL‑92 but also adds features that were not available in that version of the standard, such as windowing based on units of time. If you're not familiar with the term, windowing allows you to group records based on buckets of time. But it's not like the SQL standard was frozen in a piece of amber like some ancient insect in Jurassic Park. In 2003, a new version of the standard was released that first defined syntax for windowing functions in SQL. More versions of the standard have been released, with nine versions being released overall, and the most recent one being the SQL 2016 standard, which adds 44 new optional features, such as support for JSON parsing. This is emblematic of how the standard has progressed. As time goes on, it's more about niche features and expanding the total possible feature set than changes to the core of SQL. MySQL and Postgres both implement some features in the latest version, but often have many variations or things that they don't implement. In my experience, Postgres tends to follow the standards a little bit closer, whereas MySQL seems to be out there doing its own thing. Finally, ksqlDB borrows a lot of the grammar of Presto, a querying language developed at Facebook for querying big data on Hadoop. It was created as a replacement for Apache Hive, which served a similar purpose but was too slow for their needs. The key part here is that ksqlDB has similar origins as Postgres and MySQL, but has a much smaller feature set and is more targeted dealing with streaming data and large volumes of data, which is perfectly fine. What are the main differences to consider if you're coming from a relational database background? Well, the first thing is that tables means something very different in MySQL or Postgres compared to ksqlDB. In a relational database, tables are simply sets of rows and columns, and they form kind of the fundamental unit of measure for relational databases. If you're working with a relational database, you're thinking in tables and rows most of the time. In ksqlDB, a table technically also has rows and columns, but it behaves slightly differently. A table in ksqlDB is more akin to a reference table or a slowly changing dimension in analytical databases. In NoSQL, it bears a lot of similarities to a key‑value store. That is to say, it has a single row of data for a given key, and it represents the latest data or information on that key. A ksqlDB table represents the current state of affairs and not the history of things. If you've done any kind of data warehousing, this is basically your dimension tables or lookup tables. Next, we have streams. From a database perspective, you can treat these very much like transaction tables from a data warehouse. They represent an unbounded stream of new events that are added or appended but not modified. And that's one of the good ways of distinguishing the two. In ksqlDB, tables are about modifying or mutating data so that you have a list of the most recent values for a given key. This is kind of novel in the streaming world because this runs counter to the Kafka idea of immutable streams. These streams generally function as an append‑only kind of mode. This also leads us to two kinds of queries. At the end of 2019, KSQL was renamed ksqlDB, in part because they added support for pull queries, essentially being able to pull data from these constantly updated tables. Then they changed the syntax for push queries, which were originally available in KSQL, which pushed the results of one query to a new stream. They now distinguish between the two by adding to the end of push queries the keywords emit changes. And that summarizes the differences between ksqlDB and other relational databases, as well as the two types of operations you're generally going to be dealing with when you're working with ksqlDB.

Creating Your First ksqlDB Query

In this clip, we'll write our very first ksqlDB query, and then compare the different types of queries that are available. My friend has his own personal weather station, and it has a bunch of sensors. You can buy these for a couple hundred dollars, and it's constantly taking a bunch of readings and uploading them to the Internet. I honestly think it's a really neat project, and while I'm not sure what one person would do with all the data, I think that it's just, I don't know, it's fun. Now, what I want you to do is I want you to imagine that instead of just one person, you were trying to create the infrastructure to have open source weather reporting. Maybe you'd be able to help some sort of government reporting, maybe you'd be able to provide a free and open source of weather readings, and so you would have hundreds of sensors. You wouldn't have just one person with their weather station in their backyard, but you would have hundreds. Every hour, every minute, you would have thousands of readings for things like temperature, humidity, air pressure, and so on. This is very literally streaming data, by its definition, because it's coming in constantly from all over the place. And because it's weather data, you often want to do analyses right now, because I want to know is there a storm coming, is there a hurricane, is a tree going to fall over. These readings also count as events, because they are a brief record of something that happened in the past. So you might think, oh, well, we'll just store the data in Kafka. It is ideal for storing large volumes of streaming small events from a number of different sources. This is all fine and dandy until you want to do some analysis, because Kafka is not really designed for analysis, it's designed for storing and transporting data. And so what you might do is use ksqlDB. So we're going to look at some basic queries to see how we might start to manipulate this sensor data. So while there's at least a dozen different operations supported with ksqlDB, I think there's three main ones that you're going to care about the most. And so to help you understand how ksqlDB compares to a normal relational database, let's go through all three. First are what are called push queries. This is a select statement that pushes the results to either the command line interface or whatever interface you're using, or to a new stream. However, as KSQL evolved into ksqlDB, not only did they have push queries, but they added support for pull queries, and they consider this change to be so important that it was one of the reasons they changed the name of their product. Now, you might be thinking, aren't all select queries pull queries by definition, because you're pulling data from the database? That is true, but the distinction being made here is that generally a pull query is pulling data from a ksqlDB table, which acts as a lookup or reference table, and only contains the most recent values for a given key. So push queries push the data from one stream to another, and now ksqlDB can also pull data from what is very similar to a lookup table or a key value store, but are basically called ksqlDB tables. And as a result, it can act as a streaming ETL pipeline and like a regular database. Finally, we have joins. You can combine these two together. If you join a stream to a stream, you get a stream, and if you join a table to a table, you get a new table, but in my mind, what's really interesting is if you join a stream to a table, you now get an enriched stream, and enriched is just a fancy word for we added some stuff to it. And we're going to walk through some sample code for each of these different types of operations. So let's walk through what a push query might look like. First, we'll want to create a stream for our sensor readings. A stream is basically kind of a logical layer on top of an existing Kafka topic. Or, if you have the default set, ksqlDB will create the Kafka topic for you. And the first thing that you're going to do is specify the columns in that stream. And so, if you aren't using something like Schema Registry to predefine these schemas, you have to say, okay, these events are a zipcode, a sensortime, and a temperature, right? Then we'll specify all of the settings. So here we can see that the kafka\_topic that we're reading from is called readings, and additionally, we're saying that our timestamp is sensortime, the sensortime column. This is important whenever we're doing windowing or we're doing any kind of grouping by time, because if we don't specify a timestamp, it'll default to the row time in Kafka, which is usually not what we want. Usually we want the event time, not the ingestion time. Much like a regular SQL query, the first thing that we need to think about whenever we want to read from the stream is where we're reading the data from, so we're specifying the name of that stream that we just created. Then we need to specify which columns we care about. Here we're pulling the zipcode, and then we're taking the start of our window of time that we're going to create, and we're converting it into a string that we can read. We're saying, okay, take the beginning of our window of time, and then give me the hour, minute, and seconds. Finally, we're taking the average of the temperature. So if there's multiple readings that fall within the same window or bucket of time, we're going to average those together. Next, we specify what type of window we're dealing with. Here we're using something called a tumbling window. We're going to define these more closely later on in the course, but a tumbling window is basically a set of non‑overlapping windows. You can imagine that you want to break all of the events into our buckets and you don't want any overlaps. You just want to know what was the average temperature from 7:00 P.M. to 7:59.99 P.M. You want that all to go into a single bucket, and so here we're specifying the size of the buckets is 1 hour, and we're saying tumbling, so there's no overlaps at all. Next, we're going to group by zipcode, because we don't want everything to get mushed together just by time. I want to know what's the average temperature at 7:00 P.M. in Pittsburgh, Pennsylvania, for example, And then finally, because this is a push query, we have to add EMIT CHANGES, and those changes are either going to be emitted to the command line interface that you're running the query from or if you've specified the name of a new stream to that stream, written as new events. So now let's talk about a pull query. Let's imagine that we have an aggregated query similar to before, but instead of taking the average, we want to get the highest and lowest temperatures for a given zip code. So we might write something like this. And we can imagine that not only is this useful information in general, but that we really only care about the current value, not the intermediate values. We just want to know for all of time, what is now the minimum and maximum temperature. And so this makes it ideal for a ksqlDB table, and all we have to do is add a CREATE TABLE clause at the top, and this creates a reference or lookup table that we can query directly, or we can join against to enrich our streams. So then to make a pull query, we just write a standard SQL query. Here we're also using a WHERE clause to specify what zip code we want to read against. So now let's bring all of this together. We can imagine a situation where we want to take our new sensor data and enrich it with reference information from historical readings. And specifically, we want to know is it a high temperatures, is it a low temperature, we want to add some context. So what we can do is we can take a new stream of temperature readings and join it against our highs and lows table based on zip code. And I just want to know for each and every individual event, is this a high or low temperature? Then what we can do is we can add some logic in the SELECT statement. So here I'm adding something called a case clause, and it's basically just a big if‑then statement. I'm just saying that if it's within 5 degrees of the lowest temperature ever, it's a low temperature, if its within 5 of the highest temperature ever, it's a high temperature, otherwise it's normal. And that's it, we can run this like anything else. And here we have at the end EMIT CHANGES, so if we run this by hand, we would see the results put to the command line interface, or if we added the CREATE STREAM clause, we could push these out to a new stream. So to recap, with ksqlDB we can both modify and summarize data by using a flavor of the SQL language. Because Kafka data is immutable, the way we modify data is by creating a new stream of immutable data. We modify by appending. As for summarizing, we have two ways to approach that. We can summarize in our new stream of data by using aggregate functions like average, or we can aggregate the data into reference tables like we did with our highs and lows table.

Interactive Versus Headless Deployments

In this clip, we'll talk about interactive deployments versus headless deployments for running our queries. When we're talking about these two different modes, it can sound a little complicated or jargony. But in truth, headless mode just means that there's no user front end. There's no talking head, so to speak, for you to interact with. And headless mode, or application mode as it's sometimes called, is ideal for production workloads because you run exactly the queries you want, and you don't have someone else logging in and running a very intensive query and slowing down your results. Before we talk about the two ksqlDB deployment options, it would help to talk briefly about Kafka Streams in general. This is because ksqlDB is built on Kafka Streams, and it follows a similar architecture. The way that you deploy Kafka Streams is by deploying an entire application of program and then scaling it out horizontally so that you may have multiple instances of the same program running. When these instances talk to Kafka, they use the consumer groups protocol, which allows multiple application instances to act as one logical instance so that Kafka just sees it as one shared reader instead of a bunch of different readers. Now what if you wanted to deploy to ksqlDB? Well, ksqlDB functions very similarly. It uses Kafka Streams, but you aren't writing any Java code at that point. You're writing SQL, which gets compiled to a Kafka Streams application, and you aren't deploying applications independently. Instead, the ksqlDB Server will run them internally. This is ideal for production workloads. The way that you deploy your queries is you put them into a queries file, and when you start up ksqlDB, you specify where that file is, and it'll run just those queries. The other option is to use interactive mode, which generally means that there's a human typing code on the fly or ad hoc. Maybe a dog if they're really good at typing, but usually it's humans. Interactive mode is ideal for data exploration, ad hoc queries, and initial development. If you've worked with SQL before, you should be used to this kind of data work. There are three ways to run your interactive queries. You can use the web interface provided by the ksqlDB Server, you can use the command line client, or you can write your own requests directly against the REST API. So let's summarize these two deployment modes. First, you have headless mode, which is driven by a file with all the queries that you want to run. This leads to more predictable workloads because you know exactly what you're running, and you can measure how heavy of a workload that is in practice, which makes it ideal for production use. On the other hand, you have interactive deployments where the deployment is driven by user interaction and user queries. This means that the workload is entirely user defined, and so it can vary quite a bit based on how heavily the server is being used. This mode is great for development environments because you can run a query, see the results immediately, and then tweak it. You don't have to write a SQL file, start up a cluster, and then restart it just to make changes.

Demo: Interactively Using the CLI

In this demo, we'll start up the ksqlDB command line interface to allow us to interactively run commands against the ksqlDB server and ultimately against the underlying Kafka data. We'll look at creating a push query where we take data from one stream and push it to another stream. We'll create a pull query to keep track of the latest values. Finally, we'll combine the results into a query that tells us how the current values differ from historical highs and lows. So let's take a look at running some queries interactively against the ksqlDB server. What I have here is Visual Studio Code. And in the top section, I have a SQL file of all the queries that we're going to run. And at the bottom, I've used PowerShell and Docker to open up the ksql command line interface. So the very first thing that we want to do is we want to set auto.offset.reset to earliest. Now what this is going to do is it's going to make it so our queries will start at the very beginning of a topic and read from the start instead of wherever we are at that point in time. And that's useful for when you're doing demos and tests because you can insert some data ahead of time and then run a query. So, one of the things we can do is we can say SHOW ALL TOPICS to see what Kafka topics already exist. And right now, there really isn't anything. There's two con fluent topics that really you're not supposed to be messing around with, and then a ksql processing log, and that's it. Now one of the cool things is we can create a stream, and what it will do is it'll create the topic for us based on one of the auto create settings. So now we can see that our Kafka topic of readings exists in Kafka. We can also see the stream. Now we're going to have to scroll up a little bit. But here we can see the stream that we made called tempReadings and how it has some column information along with it. Okay, awesome. So now what we can do is we can insert some data into that stream and underlying it into that topic. And this code is going to look extremely like any kind of INSERT statement in SQL. And what I'm doing is I'm telling it, okay, I want to use the UNIX\_TIMESTAMP function to put in the current time, and then I want to put in a reading for an hour from now as well. All that math that's going on is because it's measured in milliseconds. So now what we can do is we can run this query, which is a push query, and it's going to give us the average temperature for every bucket of 1 hour. And so we can see that what it did is it gave us two buckets, and we have a row count of two rows for the first hour bucket. And the average temperature there is 45 degrees. So that's how you would work with streams. Well, what if we want to work with tables? So let's go ahead and close this. Well, I can create a table by taking a SELECT statement and just adding the CREATE TABLE part in front of it. And so now I can query that table to get the highest and lowest temperatures that have ever occurred for that Kafka topic. So here we can see that so far the highest temperature has been 60 degrees, and the lowest has been 40. But now if we enter another reading where it says 70 degrees, we can query it again with a pool query and see that it's changed. So now the max temperature for that table, for that key of zipcode, is 70 degrees. Now, I will admit this is very US centric. All of you who are using a normal measure of temperature are thinking 7 degrees, I would die in that temperature, but it's what I know. And so, here we are. Now we've done streams. We've done tables, so let's real quick do a join. So I'm going to create another stream. And this is necessary because if you were to try to join that initial stream against the table that we made, that's reading off of that initial stream, you would get an invalid topology error. You can't create these loops. So what I'm going to do is I'm going to create a second stream. You can imagine that this is new readings that we want to enrich. And so we've made the stream. And so now we can create another push query where what we're doing is we're taking the original temperature reading of this new stream, and we're adding some business logic. We're saying, okay, if it's close to the lowest temperature we've ever had, which is 40 degrees, then it's a low temperature. If it's close to the highest we've ever had, which is 70 degrees, it's a high temperature, otherwise it's normal. So now here's where it gets a little bit interesting. What I'm going to need to do is I'm going to have to split this window. And then I'm going to run this Docker command to open up another console for the command line interface. And so here's the part where it gets interesting. What I'm going to do is I'm going to insert these three records on the right, and we'll see, oh, let's make sure it's selected, I'm going to insert these three records on the right, and what we're going to see is over on the left, it will omit, it will output that enriched stream or the enriched results. And you can see how it's given us the original temperature from the new events, as well as our classification, whether it's low, normal, or high. And while it's neat that it's pushing these back to us, what we would normally do in a more production setting is we'd have them push it to a different stream, and as a result, a different Kafka topic. So that's pretty much it. It's very, very easy to get started with running queries directly against ksqlDB.

Deploying a Streaming ETL Pipeline in Headless Mode

You know how to write KSQL and run queries in interactive mode, but what would it look like to deploy an ETL pipeline in headless mode? One of the big challenges I ran into while creating this course was trying to nail down exactly what some of these terms really mean. At least personally, I find that some of the streaming data terms can be a bit confusing, or at least sound fancier than they actually are. This is especially true when we talk about the final product that you're creating. In this course, we're going to be talking about two different types of streaming applications or end products that you might produce with either ksqlDB or Kafka Streams. The first one is a streaming ETL pipeline. This sounds really fancy and really complicated. All it is, though, is some process that takes in streaming events, manipulates or transforms them in some way to new events through multiple different transformation steps, and then outputs those new events. If you've been following along, then you already know how to do this. And in the rest of the clip, we'll talk about how you would deploy one of these using ksqlDB in headless mode. The other type of end product that you might create is an event‑driven microservice, and this sounds like a totally different type of program, but it's largely the same. An event‑driven microservice is a program that communicates primarily by reading and writing events. And as a result, it's able to be small, agile, and independent of the rest of the infrastructure because it's only taking a dependency on Kafka or whatever event store you might use. It doesn't have to integrate with any other applications. Another way of thinking about this is that an ETL pipeline will transform data, and an event‑driven microservice will trigger side effects. Now those might be direct side effects like the application sending an email alert based on a high temperature warning. Or they might be indirect side effects such as a fraud alert or fraud event being put onto Kafka and then processed further down the stream. So the question for this clip is how do we implement an ETL pipeline with ksql? Well, there's only four things you need to do. First, you're going to need to look into Kafka Connect. Kafka Connect is a piece of software that allows for importing data from external data sources, as well as exporting it to external data syncs. Unless all of your data lives in Kafka for forever and for all time, you'll want to get Kafka connectors set up for the extract and load portions of extract, transform, and load. KsqlDB can run these connectors in something called embedded mode or use an existing Kafka Connect cluster. Next, you'll want to create a queries file. This is a .sql file that has all the streams and tables you want to create, similar to what we've done earlier in the course. Once you've created the file, you'll need to tell the ksqlDB server to run it. Now there's two ways to do that. The first is use a command line parameter, specifically ‑‑queries‑file, and then the file location of your .sql file. The other option is to set the ksql.queries.file property in the config file. Now, if both of these are set, the command line option takes precedence. Finally, you'll need a Kafka client of some sort to check your results because when you enable headless or application mode, it disables interactive mode. This means that if you want to interactively check the results with ksqlDB, you'll need a second server set to interactive mode. So what if you want to learn more about this? Well, there's two resources I can recommend. First is a course on Pluralsight called Kafka Connect Fundamentals by Bogdan Sucaciu. He has a very detailed course, which is helpful in understanding the extract and load portions of the ETL process. Additionally, there's a streaming ETL tutorial by Confluent. It's a hands‑on lab that covers pulling data from Postgres, modifying with ksqlDB, and then saving it to MongoDB.

Summary

In this module, we saw how ksqlDB uses its own flavor of SQL, based on the 1992 ANSI standards. This puts it on the same family tree as MySQL and Postgres, but with a simpler syntax and a few extras. We looked at the two main types of queries in ksqlDB. Our push query is where we take data from one stream, modify it in some way, and then push it to another stream. Second, are pool queries, where we take a ksqlDB table, which acts as a reference table or key value store, and we pull data from it. We saw the two ways to deploy a query. First, is to run it interactively, or by hand. The other way, which is ideal for production, is to configure ksqlDB to run a file that you've created at startup.

Understanding Advanced Stream Processing

Managing State

In this module, we're going to look at some of the more complicated aspects of working with streaming data. Specifically, we're going to focus on three main topics. First, we'll talk about the difference between a stateless application and a stateful applications. This is important because stateful applications are harder to manage and scale. And when you're doing aggregations like a sum, a count or an average, you're going to have to deal with a certain amount of state. Next, we're going to talk about how you deal with time, how you can window over time, grouping your events into buckets, but also how there's different timestamps to be aware of, both event time, ingestion time, and processing time. Finally, we'll talk about one of the biggest challenges with the distributed streaming solution, and that's how you maintain consistency across your application and avoid either double counting data or ignoring data. A term that comes up when you start talking about things like containers, microservices, distributed applications, and streaming data is state and statefulness. And it's a bad word. I don't know who picked it. It's bad because it's very generic. It has multiple meanings, so let's work through it for a more intuitive understanding. So imagine a light switch. That light switch can be on, or it could be off. Those are the two states of being that it can be in. And each of those states is essentially storing a single bit of data, 1 for on and 0 for off. So far, so good, right? So what's wrong with having state information? What's the problem with this information? Let's imagine that instead of a light switch, we have you, a programmer. And in your head, you can only hold one piece of information at any one time. And specifically, you're trying to configure ksqlDB to point to Kafka because you just watched this amazing course on it, and you need to remember what the Kafka port is. You just looked it up. It's 9092. It can't be that hard. And then, your coworker shows up, and they ask you, what's your favorite type of dog? And you say, well, I sure do love me some chihuahuas, although they are a really, really anxious breed, so they're just kind of jittery all the time. And your coworker leaves satisfied with your deep insight into canines. Now, here's the question. What was that port number? Unless you have it memorized, which if you work with Kafka, you might, or you wrote it down, it's just gone now, poof! And that's the thing. Stateless applications do work right as it comes in, and then they can completely forget about it, and that's fine. They don't have to remember what they were doing earlier or keep information in their heads between tasks. Stateless applications can scale really easily and are easy to start up. But you, you are not a stateless application. You're a stateful one, and you need to be able to keep information in your head. Let's take another example, and this one's a little bit silly. Imagine you're working on that config file, but then an alien from outer space comes by and zaps you with a plasma ray, and then they just they scatter off. Well, thankfully, this is clearly some sci‑fi world, so we just pop out a clone of you. Say, hey, you're programmer, B247. Get to work. But, because you had some state information in your head, we have to figure out a good way to share it between the you that got zapped and your clones. And in computing terms, this is called a state store. So to recap what we've covered so far, state is information that needs to be persisted between tasks. This is important because it makes scaling out and recovering more difficult because we have to configure, store or recover that state information. Tools like ksqlDB and Kafka Streams will use a state store like RocksDB to hold a local copy of that information. And they have processes for sharing that information between application instances in case of some sort of failure. Some data transformations are stateful and require keeping the data around for an arbitrary amount of time, and some are stateless and just respond to tasks or requests as they come in. And let's talk a little bit more about the distinction to make it crystal clear. Let's go back to our car analogy from much earlier in the course. Let's imagine that we had an automated carwash station like you probably see at some gas stations. Well, we might have a line or a queue of cars, and they would all go in one at a time, all grungy and dirty, and then they would get washed, and they would come out all shiny and new. This is similar to a stateless application because it's able to work on one car at a time. This would be like a ksqlDB or Kafka Streams application that modifies individual events, maybe adding some information to it, and pushes them to a new stream as they come in. So there's no aggregations. There's no windowing. It's just maybe doing a filter or some kind of mapping. Alternatively, let's take our cars again, but imagine that we have a bus station like much earlier in the course. And like before, we have to load our cars and passengers in to wait until the bus is ready. Now this part here is key. This loading and waiting is the state information. The passengers waiting are the state information. We have to have somewhere for them to either stand or sit until everyone can get on the bus, and then we can send them on their way. This is an analogy for a stateful application because you can see our little station is a lot bigger than the previous one because we have to store more stuff in it. So what kind of operations are stateful? Aggregations of any time when you're doing sums, averages, counts, and so on. Now, often you don't need to keep everything. You don't need all of the historical data, but the moment your calculation depends on multiple events, you have to keep some of that information around between reading and processing those events until you've decided that, okay, we have enough, we can output it. In the next clip, we're going to talk about dealing with time and about windowing your data, which means to aggregate your data into buckets of time. And as you can imagine, you have to keep around that information until that window of time has closed and has been finalized. Next, our tables in ksqlDB and Kafka Streams. Now remember, these tables are special because they function as lookup or reference tables and save us from having to read an entire Kafka topic every time we want to get the current values for a specific key. Literally, the entire point of these tables is to keep around data between new events coming in. Finally, stream to stream joins also require holding onto state information. These types of joins are actually windowed by time, and so the bigger that window is, it's called a sliding window, in this case, the more data has to be kept around in a buffer. So these are all things that you need to be mindful of when you're writing your queries.

Managing Time

In this section, we're going to talk about how we deal with time in streaming data. The whole idea of streaming data is that it is data in motion. It is data that is streaming, and we care about its place in the stream, its place in time. And so streaming technology appreciates the value of time. One of the biggest things that's confusing about when we talk about time and timestamps is that there are actually three different times that are relevant to the lifespan of an event. First, is when the event is born or created. This is usually the timestamp that you care about most when you're grouping and windowing by time. Next, is the ingestion time, which sounds like you're eating a sandwich, but really, this is when the data is received by Kafka. This is not guaranteed to correlate to the event time. There could be events that are dramatically late or delayed by network issues. You also have to contend with internet devices that are occasionally offline, and upload their data in batches. Finally, we have processing time, and this is interesting, because this is when Kafka Streams or ksqlDB processes the data and usually, because they're dealing with streaming data, we want us to be as close as possible to the ingestion time, but technically speaking, if you set these applications to read from the beginning of a Kafka topic, it may be doing the processing days or even weeks later. It's not guaranteed to be close to the ingestion time. So one of the key things is that now that we understand these different types of time, we're generally going to want to be able to window time, and windowing is actually extremely simple in concept. Windowing is just taking all the events in a period of time, or a window, and grouping them together. That's it. Windows can overlap or be completely distinct, depending on what type of window you use, but all of it is just another way of grouping your data together. So, let's discuss the four types of windows that are available. The first is a tumbling window. Imagine that you have a bunch of events arranged by event time, where each of these slots represents, say, a minute of time, and we can make a window that is 2 minutes long. For example, these two events would get grouped together into one window, and their data would be aggregated together. And then, if we have a tumbling window, our window would tumble or roll over, creating a new, non‑overlapping window that is also 2 minutes long. The end result of this process is that each time period, each slot here, would fall under exactly one window. All of these windows would be non‑overlapping. But, what if we want to consider overlaps? Well, we might just decide to use a hopping window. Just like before, we decide how wide we want our window to be, say 2 minutes, but we also determine how wide we want it to hop. So in this example, our hop is 1 minute. The end result would be seven overlapping windows, and here, I have them staggered a bit, so you can actually see all of them, because it would be a jumbled mess, if we had them all properly overlapping. And what this means is that an event is going to be counted twice in our example, and possibly multiple times, depending on your settings. Next, is a type of windowing that I've never seen before, and that's session windowing. Session windowing is interesting, because you define a period of inactivity, or a lack of events. So maybe your inactivity duration is 2 minutes, and what the system will do is it will look for a gap in your event stream, and then the events between those periods of inactivity, those gaps, the events between are grouped together into dynamically sized windows. And this makes a lot of sense when you're talking about a stream of user interaction events, like if someone is shopping at your online store, and each of these events is a click, for example. So this is really cool. Finally, we have sliding windows. Sliding windows only apply to stream‑to‑stream joins. These are only for doing joins. So imagine, if instead of one event stream, we had two, and we said that if any events happened within the same 3 seconds, we want to join them. Well, this window will slide along the timeline, and if two events from the different streams fall inside this window at the same time, they will be joined together. In practice, though, it isn't just a free‑for‑all like this. Usually, you're joining on a key, and the sliding window helps control how big your memory buffer gets. This prevents the system from having to compare the entirety of one stream to the entirety of another stream, just to do a join. Joined events have to be somewhat close in time. When we're dealing with windowing, we have to talk about terms like grace periods and retention periods. Grace periods are important, because the event time can differ wildly from the ingestion or processing times. If an event gets delayed, it will arrive in the stream out of order and late. The grace period is how long we're willing to wait for these late events to show up. In the example with the bus, it's how long we're willing to have the bus wait around for everyone to get on before we leave. And the thing is, we're not going to delay the bus for an entire day just because somebody didn't show up. There's the retention period, which is how long the system will keep those events at all, or how long it will keep the windowed results, if you're doing something like a table. This value should be bigger than the length of the window, and the grace period combined, which makes sense, because you don't want to delete events before you're done with them.

Managing Consistency

Let's talk a bit about managing consistency with streaming data. So what might be considered a consistency failure? Well, first is something that you could call read never. Essentially, because of a failure, an event never gets read and never gets processed. On the other end of things is read multiple, and you can imagine this being a problem for some types of streaming applications, but not others. If an event gets duplicated or read twice, it might alter a calculation, or maybe cause an alert to go off multiple times. This would be really bad if we were talking about something involving money, like ordering the same product twice. But for something like a personal weather station, a few duplicate readings would be perfectly fine. We really wouldn't care about the slight change in outcome. The ideal would be read once and only once. So the first question is, how do we avoid a read never scenario? Ideally, we want some sort of state store that can be migrated or reassigned to another processor if one of them fails. In Kafka Streams, you have granular control over what type of state store you use, such as RocksDB. But in addition to that, you need a way to rebuild that state store if you haven't told the system to replicate copies of it to the other processors. For ksqlDB and Kafka Streams, this means reading from a compacted change log topic that tracks all of the state changes for a long as they're relevant. Additionally, as part of the recovery process, the work needs to be assigned to that other processor so it can finish it. KsqlDB and Kafka Streams both handle this for you automatically. So that's how we're able to avoid a read never scenario because we'll just restart up to work if we need to. But what about the read multiple scenario? How do we avoid double counting or double producing our events? Before I can give you the answer, we need to define a word, and that word is idempotent. Idempotent is a fancy word from mathematics, and it applies originally to mathematical operations, like taking the absolute value of something. If I give you ‑7 and say take the absolute value of it, it's positive 7. But then, if I take that and say do it again, well the absolute value of positive 7 is still positive 7. Idempotent operations are ones that if you repeat them over and over again, you get the same result. But when we start applying this to computer science, a better way of thinking of it is an action that is safe to repeat. For example, sending a get request to a Wikipedia page is safe to repeat. You just get the page multiple times. But hitting post on an online forum using a post request in HTTP is not safe to repeat because you'll end up with a duplicate post. This is why, when you try to hit the back button sometimes on your browser, it'll warn you, hey, you can't do that. Specifically in the event streaming world idempotency means that if your system receives the same event multiple times, it can be handled properly. This can be handled through duplicate detection, unique IDs, or sequence IDs, depending on the scenario. So why do we need to know this word? We can actually choose which of these scenarios we want ksqlDB or Kafka Streams to operate in, and all we have to do is configure the processing guarantee setting. Specifically, the two options are called exactly‑once and at‑least‑once, and they work pretty much the way that they sound. At‑least‑once is the default for ksqlDB in Kafka Streams. But why isn't it exactly‑once? That's better. Why would we allow duplicate events to be produced? Why wouldn't we always want it to be set to exactly‑once? Well, the thing is, exactly‑once semantics, or exactly‑once processing guarantees aren't free. They come with a small performance and storage cost. Your processor, in this case, Kafka Streams or ksqlDB, needs to be set as idempotent, the word we just talked about, in Kafka. This means that Kafka knows to get rid of any duplicate events that have been produced multiple times by using a sequence number that's added to the event, similar to how TCP works for the TCP protocol. Now, this will increase the size of your events by a small amount because we're adding information. Additionally, Kafka Streams and ksqlDB have to use the Kafka Transactions API in order to track changes to state, track changes to the topic offset, and output the transformative events atomically. Atomically means they happen all at once or not at all. This is very similar to transactions in relational databases. In short, these tools will do all of this difficult work for you, but expect a small performance overhead.

Summary

Sometimes our programs need to keep information around, depending on what they're doing, especially if they're doing aggregations or windowing, and this is called state information. We can use windowing and grace periods to aggregate over time, usually based on the event time if it's available. Finally, we can achieve exactly once processing semantics and processing guarantees, but there is some performance overhead.

Creating a Kafka Streams Application

Getting Started with Kafka Streams

Now that we've talked about ksqlDB and the theory of streaming data, let's go a layer deeper with Kafka Streams. In this module, we will walk through the steps that your program has to take if you want to start using Kafka Streams. We'll talk about some of the different types of transformations, especially the split between stateless transformations and stateful transformations. Finally, we'll look at making applications that are event driven instead of the more traditional request and response approach. Earlier in the course, I talked about a way that we can compare a nonrelational data system like Kafka and Kafka Streams to a relational one like my SQL or Postgres. At the bottom, you would have your data storage. In our case, that would be Kafka for our durable storage and a state store like RocksDB for our transient state information. Once you have the data, you need some kind of execution engine to apply the data transformations you want to make, which would be Kafka Streams, in our case. And then finally, you might add a querying language on top to make it accessible to users with experience using SQL instead of having them try to write Java code. But when we dig into Kafka Streams, it's not one single piece, but in fact, two. First we have the processor API, which is a low‑level API where we can interact with the state store and write our own custom processors. Built on top of that is the Streams DSL, or Domain Specific Language, which is what we'll be focusing on in this course. The DSL adds the abstractions of a stream or table and provides a bunch of common data transformations out of the box. I highly recommend starting with this because it's so easy to use and only then dipping down to the processor API when absolutely necessary. So what do we need to use Kafka Streams? You'll need Java and a Java integrated development environment. Alternatively, there's also a Scala version of the Kafka Streams Library. In the demos, I'll be using Eclipse, but any Java IDE should do. Additionally, you'll want Maven, since that's the easiest way to add Kafka Streams as a dependency to your project. Finally, you'll want to be including the Kafka Streams library for your project, and possibly other libraries from Apache, such as their Kafka Clients library for serializing and deserializing data. So what is actually involved with writing your first Kafka Streams application? Well, it can be hard to discern just from trying to look at the code, but in fact it's actually pretty straightforward. First, you'll need to import any libraries which you're using, which is pretty standard for any kind of Java programming. Next, you're going to have to configure some settings. At a bare minimum, you have to tell your program where to look for the Kafka broker and preferably give it a couple of options, in case one of those places is down. Next, you specify where you're pulling your data from. And then, the part we care about most is actually making the data transformation. Finally, if we're writing to a stream, we'll want to export that data back to Kafka. So once you've used Maven to add the Kafka Streams library as a dependency to your project, you'll use the import statement to pull in any classes that you need from that library. Additionally, you'll be creating a properties object and then setting the properties for your project. Now, there is one specific property that we need to talk about, and this is called a SerDes, which is actually just a made up word. SerDes is an abbreviation or combination of serializing and deserializing, so, ser, serialize, and de, deserialize. Essentially, when we write data to Kafka, there are a multitude of ways that we can read and write those bits and bytes. And initially, Kafka Streams has absolutely no idea which one we want. So we need to specify what type of SerDes, or serializer and deserializer, we want. Sometimes that will be simple data types like text strings and integers. Other times, you might be using a more complex data format like JSON or Avro. To better explain things, let's imagine you have a favorite number. If you're anything like me, that number is probably 18,537. You look at 18,537 and think, that's such a good number. I want to put it on Kafka and store it. But Kafka doesn't store numbers per se; it stores binary. So what you would do is, you would use an integer SerDes, which is going to break up your big number into two smaller numbers. So, for example, it might break it up into 72\*256 + 105\*1. Think of it like writing a number in base 256 instead of base 10, like we do for decimal numbers. And then it would convert each of those smaller numbers into a byte in binary, which can hold up to 256. And so now your friend Susan, who also works with you, is going to read your favorite number from Kafka. Oh, but wait, you accidentally told her to use a text string SerDes. Oh, geez, this is probably going to be totaled gibberish. So the program reads the two bytes, and it turns them into letters. In UTF 8, the number 72 converts to an uppercase H and the number 105 converts to a lowercase i. This is terrible. Now she thinks you're trying to use Kafka as the world's most awkward texting application, when really you just wanted to share your favorite number ever. Once we've imported our libraries and specified our main settings, we need to create a streams object or do something similar with a table object. As is common in Java, we'll be using the builder pattern. In this case, we're going to define the topology of our data flow by indicating which topics we're reading from with the stream function and where we're writing to with the to function. Normally, we'll also be doing some transformations in between. Once we've defined our data flow, we can build it to create our topology, or the shape of our data flow. Finally, we can input our properties and topology to instantiate a KafkaStreams object, which is the part that actually does all of the work for our program. What's important here is we don't want to just shuffle around data from one topic to another, and so we want to be able to apply some transformations. So you can imagine that we might create a KStream object, and then based on that stream, create another one but with some transformations applied. And so here we're calling the filter function, and we're passing in an anonymous function, a lambda function, as it's called. And all we're saying here is that okay, if you give me the key and the value, I'm going to evaluate whether that value's between ‑50 and 130, and I'm going to return true or false. If it's true, then it's going to get passed to our new stream and eventually to a Kafka topic, and if it's false, we're just going to drop it. And let's talk a little bit more about some of the types of transformations that are available.

Transforming Data with Kafka Streams

There's a number of transformations that you can use whenever you're working with Kafka Streams. The types of data transformations available largely fall into two main buckets. First are stateless transformations, which are transformations that generally work one event at a time and don't have to retain state information between event processing. As we covered earlier in the course, stateless transformations require less RAM, can be more responsive, scale up better, and are more quickly able to recover from failure. Stateless transformations are the ideal. The other type, unsurprisingly, are stateful transformations, transformations where we have to keep some information around between events. Anytime you're looking to do an aggregation or windowing, you need state information. So what do you think of when you think of stateless operators? Well, a common type of operator in this example is filtering or directing the data. A simple filter discards data based on some condition, or you could do an inverse filter, which keeps the data based on some condition. A branch will direct events to multiple possible output streams based on a specified predicate or condition. If none of the conditions are met, then the event will be dropped. It's apparent when you think about this that you can do this work one event at a time, without reference to any of the other events. The rule of thumb is that if you can process an event and forget it, it's a stateless operation. Another common operation is the map operation, not like the map of a country, but like mapping from one item to another. This gets into the idea of mathematical functions and how they can be thought of as mapping from inputs to outputs. So, for example, if you want to take this square of a number, 1 maps to 1, 2 maps to 4, and 3 maps to 9, and so on. In programming, this generally just means taking a series of input events and applying or mapping the function to each and every one of them, one at a time, and then outputting the results to a new stream. The flatmap function is similar, but is designed for functions that might return 0, 1 or more events. And so the multiple results need to be flattened back out into separate events. So back to our math example, if we were to have a square root function, the square root of 1 would be 1, or it could be the imaginary number i, if, for example, we were allowing complex numbers. So, in that case, we would use the flatmap function to return a flattened list of the results into separate results. Finally, are the GroupBy and GroupByKey functions. When you do a GroupBy operation, you are inputting a KStream and outputting a KGroupedStream object, which is then necessary for doing any types of aggregations. Behind the scenes, KafkaStreams will create an internal rearrangement topic if it's necessary to repartition the data. If the key for your input and output topics are the same, I recommend using the GroupByKey function so you can avoid that from happening. And then we have stateful operations. The first and most common example is any kind of aggregation such as count, sum, min, max, and so on. The moment you need to retain information from previous events, you are now in a stateful mode. Another item is windowing, which is a kind of grouping based on windows of time. This also requires keeping track of what data is in the window until the grace period for late events has expired and the window is closed. Finally, joins require a buffer of sorts to implement something called a sliding window. When you're joining two streams together, you're basically joining on matching events within a window of time, and so you have to keep data around for that window of time.

Demo: Creating an ETL Pipeline

In this demo, we'll look at some of the basics of configuring your project. We'll write some code to transform a stream of data, and then we'll use ksqlDB to easily test that program and confirm the results. We're going to create the most simple Kafka Streams application possible, to give you a foundation to build more complex applications with. Now, what we're looking at here is the Eclipse IDE for Java development, although you can use it for a number of programming languages. And specifically, the file we have open here is the pom.xml. POM stands for Project Object Model, and it's used by Maven to be able to manage our dependencies, and we can see here that we have two dependencies that are really important. The first one is Kafka Streams. This is the library we're using to interact with Kafka in general, and create our streams and table objects. Second is Kafka Clients. This is important for our serialization and deserialization, or SerDes, for the data that we want to work with. We're able to use the Kafka formatting, because we've included this library. So let's go ahead and take a look at some actual code. So here we have a very, very simple Kafka Streams application, and the first thing we do is we import all the classes that we know we're going to use. This is extremely common any time that you're working with Java. Next, if we scroll down, we've got our class, and we've got our main function, which is what gets called when we run the application. At the top here, we have the APPLICATION\_ID, which is how Kafka Streams will identify itself to the Kafka broker. We have the BOOTSTRAP\_SERVERS, so it knows where to look to connect to Kafka. And it's important that you get this right, because I literally spent probably about 30 minutes, pulling my hair out, trying to figure out why this wasn't working. And specifically, you'll notice there that we're using the port number of 29092, instead of the usual 9092. Now it's up to you whatever you want to use, but if you're using the quick‑start docker‑compose file that we used earlier in the course and was provided by Confluent, you'll notice that port 29092 is the one that's exposed to the rest of your computer. And so if we want to talk to our container that has the Kafka broker, that's the port that we would want to use. Next, we're specifying the SerDes classes. Now, as I said earlier in the course, SerDes is a made‑up word. It stands for serialization and deserialization. And what's going on here is we're telling it that if we don't specify how to serialize or deserialize our keys and values, then default to the string type for keys, and the in\teger type for values. In production, your values are going to be far more complicated, but for something like a simple demo, this works great. So now if we scroll down, we can see where the magic happens. What we're doing here is we're creating a StreamsBuilder, which is a common pattern when you're dealing with Java. And then, based on that, we're reading from the RawTempReadings topic and creating a stream called rawReadings, and we're specifying that it has a key value of string, and a value value, or value type more accurately, of integer. Then we're doing the same thing for validated readings, but instead of basing it off of a specific Kafka topic, we're taking the previous stream. And so we're creating an ETL pipeline, because we have multiple chained steps. And here, this is the important part, we're applying the filter function, we're applying the filter operation. And specifically, we're using a feature in Java 8 or later, called a Lambda, which is an anonymous function. All you really need to know is that we're telling it, on the fly, here's our filter function. It's a function with no name, that's why it's anonymous, that takes in a key and a value, and then it returns the result of this expression: value is greater than ‑50, and value is less than 130. So it's going to return true or false. Now what are we doing there? Why are we doing that? Well, I am being a little bit American‑centric by using Fahrenheit temperatures, but I promise in the next demo, we're going to fix that by converting things to Celsius. However, ‑50 degrees in Fahrenheit is extremely cold, and 130 Fahrenheit is extremely warm. And you can imagine a situation where you have a bunch of raw readings, and you want to validate that they're actually physically possible or physically likely. And so what we're doing here is we're saying, if it's not between ‑50 and 130, just drop it, filter it out. Now, in a more complex program, you might use something like the branch function to push those erroneous readings to another topic, so that you can review them later. But once we've applied our filter, we then use the to function to write it back to our ValidatedTempReadings topic. So that's all of the legwork that we really care about. The rest of this is just some boilerplate. We create our topology using the builder.build function. Now that we're done defining the different steps involved, we're going to print out topology.describe, so we can see kind of the shape of our data flow in a text‑based format, but you really don't have to do this normally. Then, we're going to create our KafkaStreams object. This is the brains of the operation, and we're going to pass in our topology and our properties. Then we're going to call cleanUp, which cleans out all of the state. Normally, you're not going to want to do this in a production environment, but because this is a demo, this is a nice way that if you have a crash or an error or a problem, you can make sure that you're starting from a clean slate. Then we tell it start, and finally, we're adding a shutdown hook. So, if we manually close the program, it's still going to be able to do some cleanup. And so specifically, it's going to call the close function on our streams object. So with all that said, let's go ahead and run this, and we can expand it up, and see that we've got our topology. It gives us our source, our sync, and any processing steps in between, and we can see quite clearly that there's the filter step that's going on. We can also see the topic that we read from, and the topic that we sync to. So this is great and all, but how do we make sure it's working? Well, what I've done here is I've written some SQL code to run using ksqlDB, and so what we're able to do is we can make sure that we create our topic, and we're going to do this implicitly by creating a stream, a different stream, in ksqlDB. So I'm going to go ahead and run this. And the key thing I want you to notice here is one, we're specifying a key value with sensorID, which is important for most things you're going to do with Kafka Streams, because usually it's assuming there's some kind of key. The other thing I want you to notice is that for the key format and the value format, we're using Kafka, and because we're using that Kafka Clients library, this allows us to easily read and write between ksqlDB and Kafka Streams. For a more complex application, you might use JSON or Avro or Protobuf, but in this case, because we have such a simple demo, this is perfect. Now, we're going to create another Kafka topic implicitly for our validated readings. And here's where the fun part is. So, we've got a streaming query. We've got a push query, where we're going to read anything that ends up in the validated temperature readings. And I've gone ahead and I have split out the window, so I have a smaller KSQL window going on over here on the right, and now we can see if our Kafka Streams application is working or not. So, first, let's just do a simple insert. So it took a little bit of a second, but we can see that it came over perfectly. Now here's the fun part. We've got two temperature values here. Twenty is a little bit cold, it's below freezing in Fahrenheit, but it's fine. Two‑thousand is going to melt most things, so we know for sure that that's an erroneous value. So our Kafka Streams application should filter that out. So let's go ahead and insert both of these. And we can see that the 20 degrees Fahrenheit came over, but the 2000 didn't. So now you know how to create a Kafka Streams application and validate that it's working. As we continue on, we'll see some other ways that we can validate that it's working, and build more complex logic with Kaka Streams

Making an Event-driven Application

In this clip, we'll look at how we can move from simply transforming data to taking action based on that data. So, one of the things I want you to take away is that being event‑driven, air quotes, event‑driven, is more of a design philosophy than some kind of light switch that you turn on or off in your program. There's not a specific line of code that you can point to and say, a‑ha! Now it's an event‑driven application. So how can we compare it to a streaming ETL pipeline, or a traditional request and response application? The best definition I've been able to find for what an event‑driven application actually is, is pretty simple. In my mind, an event‑driven application is an application that communicates primarily via events. That's it. And I know that sounds like a tautology or some kind of self‑referential definition, but I find it helpful, because it's so simple. A lot of what actually makes event‑driven driven comes from the design applications whenever you make events a first‑class citizen in your program. Why have this clip at all then? How does a streaming ETL pipeline differ from an event‑driven application? Based on our definition, they're practically the same. It's just a matter of your reference point, because a streaming ETL pipeline communicates primarily by events. Now, if you're a database person, you're probably thinking in terms of ETL, and so ETL focuses on transforming data. But, if you're more of a software developer, then you're thinking in terms of applications and code, and so how an event‑driven application might differ is that you might have more complex logic than you would normally find with, say something like the SQL language. And so this is where things start to get interesting. We can write custom functions and apply them to our events. But another, more interesting thing are side‑effects. Notice that in our previous definition, we said it communicates primarily through events, but you may decide that you want to trigger some sort of external command or action. These are called side‑effects, because they get around the whole immutability thing. They change things. They do more than just receiving and publishing events. So how can we implement these differentiating factors in Kafka Streams? Creating your own custom logic is very, very easy compared to, say, ksqlDB. The first way you might do this is with the map function. This allows you to apply a function to each event, one at a time. It's a clean and simple way to add custom logic. Now, what if you need to handle multiple events? Well, you can use the reduce function, which allows you to combine multiple pre‑grouped events into a single value, if the output is supposed to be the same data type. Map and reduce should sound familiar, if you've worked with functional programming languages, or big data tools like Hadoop. Kafka Streams also has a more generalized function called aggregate, that allows you to produce a result of a different data type. Finally, if you're using the processor API, you can call the transform function on a stream to apply the custom logic that you programmed into your processor object. So what about side‑effects? How do we implement those? First, there's the peek function. In a sense, it allows your program to look at an event, but don't touch it. It allows you to run a function based on that event, but you don't make any changes to the event or event stream. This is great if you want to implement, say, some debug logging, for example. Another option is the foreach function. The difference between peek and foreach is that the later one is a dead end. It's treated as a stream sync, and no more processing is done after the foreach. Finally, if you're using the processor API, the process function works similar to the foreach function, and allows you to apply your own custom processor logic. One last thing that I want to know, side‑effects are dangerous, and not in the drug‑commercial sense, where they'll give you baldness or nausea or some other weird consequence. Side‑effects are dangerous, because one, they are unpredictable, and two, they're usually not idempotent, or safely repeatable. And so when you're using Kafka Streams, you lose out on those processing guarantees that we talked about earlier in the course. If you decide to send an email every time an event is processed, you run the risk of multiple emails being sent for the same event, if there's a failure and a rollback. So remember, try to keep the side‑effects to either a minimum, or to implement them in a way that can handle repeated reads of events.

Demo: Creating an Event-driven Application

In this demo, we'll show you how you can do more than transform data. We'll apply custom business logic to our events, and then we'll produce side effects instead of just receiving and publishing those events. So here we've got our application. We're setting up all of the configuration, but the part that we care about is right here. So we're creating a stream based on the raw temperature readings topic in Kafka. But, because I'm an American, I put all of the readings in Fahrenheit, and we want to convert that to a more standard system of Celsius. And so you can see here that we're creating a new stream called convertedReadings, and we're taking three specific steps. The part in the middle, the mapValues, is applying a values function to each and every event. The difference between mapValues and just map is that mapValues expects a function. Here we're giving it an anonymous one or a lambda that takes in a value and returns a value. If you were to use the map function, it would require taking in the key and the value and returning a new event. So here we're keeping it simple, and all we're doing is we're saying the temperature in Fahrenheit, ‑32, divided by 1.8 is our new Celsius temperature. So we're applying some really, really simple custom logic here. But you could put in almost any function that you want as long as it takes in the value and returns the value. And here, we've specified that the value type is double, but you could use something else if you have a different type of value in your event. So we've applied some really simple custom logic that we could expand later on. But the interesting part is that we have two peek calls. And so the peek function takes in a function as well. And in this case, we're using it twice. We're giving it, again, an anonymous function, so it doesn't have a name, but the function will take in a key and a value as we can see. And then it's going to run system.out.print, which is a side effect because what's going on is we're not modifying the event. We're just taking a look or a peek, and we're printing out the value before we changed it and then after. Now, thanks to the fact that I set this consumer configuration of AUTO\_OFFSET\_RESET\_CONFIG, which is a mouthful to earliest, when we run the program, it's going to read from the start of the topic. So I'm going to go ahead and run this, and what we'll see is that there are three events that I populated ahead of time. And so what it's done is it's taken in those three events. And so there was ‑40, 32, and 212 Fahrenheit, and it converted them to Celsius. ‑40 is interesting because it's the same temperature no matter what system you're using, and then 32 in Fahrenheit is freezing, and 212 in Fahrenheit is boiling. So here, we're able to see that without a lot of effort, we would apply some custom logic, as well as implement some side effects.

Summary

In this module, we looked at how to use Kafka Streams in your application and how it's really just a library to import like any other that makes it so easy to transform your data that's living in Kafka. We talked about the types of data transformations that you can apply, which tend to fall under the buckets of stateful or stateless. Finally, we looked at how we can do more than transform our data by applying more complex custom logic and by triggering side effects outside of just creating new events. In the next module, we'll cover how to take your streaming applications even further by applying joins and aggregations.

Aggregating and Enriching Data with Kafka Streams

Aggregating Data

In this module, we'll see how we can aggregate and enrich our data using Kafka Streams. Overall, this module is going to be pretty simple. First, we'll see how we can take multiple events and combine them together. Next, we'll look and see how we can use joins to enrich our streams with data from either other streams or from lookup tables. The moment you want to deal with multiple events, instead of doing them one by one in a sort of stateless fashion, you have to take a few different steps. First is grouping. This collects the events together based on a key, but is considered a stateless operation. Depending on what key you're using, Kafka Streams might create an internal repartitioning topic to repartition the data based on the new key. Because Java is strongly typed, this operation will turn a KStream object into a KGroup Stream object or a KTable object into a KGrouped Table object. Again, by itself this is stateless and efficient, but I can't imagine that you would only want to group a stream. As part of that grouping, you might define a time window. So, for example, you might want to group weather sensor data by geography or sensor ID, but then still want to create hourly windows of time so that maybe you can take the average temperature for each period of time. Finally, once you've applied any groupings or windows, you'll want to aggregate the data. You might want to use a built‑in aggregation like count or sum, or you might decide to create your own. So what do you need to do if you want to create your own aggregations, either using the reduce function or the more generalized aggregate function? Well, first, you'll need an initializer. If you were to create a sum function, for example, the initializer would be the value 0. Whenever you're using the more simple reduce function, then your initializer is implied. It's essentially the first value that comes in. Next, you'll need an adder function for when the data is added to the aggregate. So if we made a function to count how many times the letter Q was used, our initializer number would be 0 and our adder would count the number of Qs in our event and then add it to the running total. Finally, you need a subtractor. Now, this seems kind of silly. When would you ever remove data from an aggregation? You're going to read all the data, why would you get rid of anything, right? Well, windows are a perfect example where that's not the case. With windows of time, events are being added and removed all the time, and so if you're making a Q counter to count how many times letter Q comes up, you need a way to subtract from that count when the phrase quirky, quizzical Quetzalcoatl is being removed from your time window. Finally, there's some efficiency concerns to take into consideration for this kind of stuff. First is that stream‑to‑stream joins are a bit inefficient because you often have a lot of data to deal with, and as a result, you are forced to specify a window of time so that you're not joining everything in the stream versus everything in the other stream. However, in situations where you only care about the most recent values for a key, like current location for a moving object, then something like a KTable is more ideal. It is much preferable to join a stream to a table than it is to join to another stream, if possible. Next, when you're joining a stream to a stream, try to limit that window width to as short as a duration as needed. The larger the window, the more state information that has to be kept around and the slower the join will be. Finally, anytime you're making a join, your objects have to be co‑partitioned in the same way, and Kafka Streams expect you to handle this. What this means is that they have to have the same number of partitions and they have to have the same partitioning method. Basically, Kafka Streams is going to assume that you've made it so your two objects have the same partitioning scheme, and so it can just join them together without having to do a lot of work. It will confirm that they have the same number of partitions, but the rest is up to you.

Enriching and Joining Data

In this clip, we'll see how we can take raw, unaltered events and enhance and enrich them with data from other sources. So there are three types of joins that are supported by Kafka Streams. First, are inner joins, where you take two sets of data, and you only return values where there's a match on both sides for the keys. This is the most restrictive type of join, because you'll drop any data that you can't find a match for. Next are left joins, and sometimes in the SQL world called left outer joins. These keep all the data from the left side, which will usually be a stream, and bring in any data from the right side, which could be a second stream or a table. As you can imagine, this is a little bit more expansive than an inner join and will include more results. Finally, we have an outer join, and sometimes it's referred to as a full outer join in the SQL world. This includes data from both sides, even if there isn't a match. This type of join is the most expansive and has the least support in Kafka Streams. Speaking of supported joins, this is going to be a complicated slide, because there's a lot of factors to consider. You can't just do any kind of join that you want. You have to consider first, what objects you want to join together, and then what type of join you're doing. So first, there are stream‑to‑stream joins. These always have to be windowed to limit how much data is being compared at any point in time. Unsurprisingly, if you join a stream to a stream, you get another stream, and, if you join a table to a table, you get another table. Now here's the important question. What do you get if you join a stream to a table? Any ideas? Well, let's think about this for a second, and see if we can figure it out intuitively. What would be the point of joining a stream of events to a reference or lookup table? Well, it would be to enrich the original stream, and so it makes sense that the output is a new stream, the enriched one. Finally, we have something called a Global KTable object. This is exactly like before. It's exactly like a KTtable, but it's not partitioned. A Global KTable contains all of the data in that table replicated to each application instance. These are useful whenever you either want to have to avoid some of those co‑partitioning requirements, or you want to be able to speed up your joins. The downside is that it takes up more memory. So now that we've covered the object pairings that we can do, what about the join types that we talked about earlier? Well, first is an inner join, where data from both sides has to match up, or it gets dropped from the final output. Every type of pairing here is supported for inner joins, and this makes sense, because inner joins are the most restrictive, and therefore a little bit easier for Kafka Streams to handle. Now, what about left joins? Well, this gets more complicated, because what is on the left side in this case? Well, notice the order that I put the words in at the top. The left side is almost always a stream. And so in a left join, we're keeping all of the events from the stream, and possibly enriching it with data from another stream, or KTable, or Global KTable. The only exception there is the table‑to‑table join. Well, as it turns out, these are all supported as well. Now for the final question. What about an outer join, the most expansive of the three types of joins available to us? Well, in this case, you can only join an object to the same type of object. So here you can only join KStream to another KStream, or join a KTable object to another KTable object. Finally, what considerations do we have to make whenever we're doing our joins? Well, there's quite a few. First, null keys won't join. So you need to make sure that you have a key value for every event in your stream. Another thing is that if you're joining a stream to a table, only changes in the stream will trigger the join. So even if you add a key to a table, that wasn't previously there, and has validated on the stream side, this won't cause a join event to happen, and it won't output new records. Instead, you have to wait until a new event in the stream is added before a join will occur. Next, we already spoke about object types. Joining a stream to a table can be better than joining a stream to a stream sometimes, and joining to a Global KTable can allow you to ignore partitioning requirements at the cost of more memory and storage. Speaking of which, your data, like we said, has to be co‑partitioned, and Kafka Streams isn't going to do the work for you. It'll check that both sides have the same number of partitions, but in addition to making sure that that matches, you also have to make sure that the same partitioning scheme or structure has been used, so that keys are distributed to application instances in the same exact way for both objects. Finally, any time you're doing a stream‑to‑stream join, you'll have to apply a sliding window to limit the range of time that the events are compared for. You'll want this to be as small as possible to improve performance, but big enough to avoid missing any matches.

Demo: Aggregating and Joining Data

In this demo, we're going to see how we can create an aggregation. First, we're going to branch a stream into validated events and erroneous events. Then, we're going to create an aggregated table on the errors to track how often erroneous events are coming from our sensors. Finally, we will join the validated events to that table so that we can flag any possible sensors that might be producing inaccurate results and avoid further downstream from using those, even if those results are in the proper ranges. In this demo, we're going to see how we can group and join together data. Specifically, we can see at the top that we're creating the rawReadings KStream based on our RawTempReadings from our sensors. And then what we're doing is we're using the branch functionality to say okay, we want some of these to be valid readings and some of these to be erroneous readings. And so the way that the branch function works is that we give it a number of conditions or predicates. Here we're using anonymous functions, and we're saying, okay, take in a key and a value for the function, and then return true or false. In the first case, if the value is between ‑50 and 130, which is the range that we want, then it's going to go to the first branch. Otherwise, it's going to go to the next option. And in this case, it takes in a key and a value and always returns true. So what's going to happen is we're going to have an actual array of KStreams, and all of the valid entries go into the first one, and all of the erroneous entries go into the second one. So, so far, so good. Now, to make things more interesting, we're going to group that stream, that second one of erroneous value. So we're creating a variable called gs, and we're creating a KGroupedStream based off of that. We're using the groupByKey function because we're not changing the key at all. Now, we're breaking this into multiple steps, but you could always chain these together. Based off of that grouped stream, we're going to create a KTable object. And all we're doing is we're using the built‑in count aggregate because we don't want to have to make our own. And then once we do that, we're using mapValues to turn the results from the count into an integer so that it just makes everything cleaner for what we're doing because the count aggregate returns a long data type, which is much bigger than an integer. So at this point, what we have is we have our errorCount KTable, and that's what we care about. Specifically, we're going to take our validated readings, which are the first option in the branch of the filtered readings, and we're going to join it to that errorCount because what we want to do is as new events come into our validated readings, we want to make sure that a specific sensor ID has not had a large number of anomalies. In that leftJoin function, we give it the KTable we want to join against, and then we give it a value joiner function. And in our case, it's pretty simple. So what a value joiner function does is it takes in the left side, the value on the left side, and the value on the right side, and it returns a combined value. In our case, what we're doing, our values are very simple. They're both integers And what we're saying is if the right value is missing or null, so we don't have a historical record in our errorCount, or if it's less than 3, so there haven't been 3 or more errors, then, using the ternary operator, we're going to return the left value. We're going to return the validated temperature reading. However, if there have been 3 or more errors, we're going to return ‑99,999. And then finally, we're going to use the to function to write this out to the topic ValidatedTempReadings. So let's go ahead and run this program. And so we can see that we started the error counting. So let's actually insert some values and see what happens. So here we have VS Code, and I'm connected to the ksqldb Docker container. And so I've started a push query to see the results in the validated piece. And so if I go ahead and just insert these, they're going to come over the way that we expect. But if we insert an erroneous value like 5000 degrees Fahrenheit, we can see that that doesn't come in. And now what I'm going to do is I'm going to insert two more to trip our alert system. So now if we have a regular reading, because we've had enough erroneous readings in the past for this sensor ID, it's going to return a temperature of ‑99,999. You can see that what we've been able to do is we took a regular stream of data, we split it up, made a table based on one, and joined that table to the other to identify whenever our validated readings might have too many errors.

Demo: Windowing Streams

In this demo, we're going to see how we can apply windowing over time. First, we'll create a stream. We'll apply a window function to it, and as part of that, we'll specify how to handle late events. And then finally, we'll apply an aggregation over that window, to be able to see how many events came in during that period of time. Here at the top, we can see that we're using Duration to define how long our window size is going to be, and our grace period size is going to be. So our window size is 2 seconds, and our grace period is 1 second, which is how long we're willing to wait for new data. Now, if we scroll down a little bit, we can see here that we've created a stream based off of raw temperature readings. And so we're just pulling this in, and then, based on that stream, we're going to create a KTable. We're going to group by the key, and we're using groupByKey function, because we're not changing the key, so the system knows that we don't have to re‑partition anything, which is convenient. Then we're using the windowedBy function to break it into windows of time. Now, you'll notice here that for the window, we're only specifying the window size, not the distance moved each time you have a new window. And because we're doing that, because we're only passing in one parameter there, we're using a tumbling window by default. But if you wanted to use a hopping window, there's a way that you could specify how far each hop would be. We're also using the grace function to put in the size of how long we're willing to wait for late and out‑of‑order events to arrive before we close out the window and send out the result. So this produces a KTable, and it has a windowed string as its key, which means it has not just the string or the sensorID for what we're putting in, but it also has information about the window. So that's how that's handled, if you're wondering, okay, where would we fit this information? You'll notice a little bit further down, we're going to take that table and turn it back into a stream of events. So how does that work? Well, we run the toStream function on it, and then we apply a map, and this is where we extract our information about the window that we've created. So, we're using an anonymous, or Lambda function, and we're taking in a windowed string for our key, and then a long for our count or our value. And based on that, our function produces a new key value pair, and specifically it extracts that original key from that key variable, and then we're creating a string that's a combination of the start of the window, as well as the final count value. Finally, what we're doing at the very end is we're using the print function to write that all to console, just for the sake of ease. In a production environment, you would be writing this out to another stream, or keeping a table, or doing something like that. So let's go ahead and start this up. And now let's insert some data. So I've already created the stream here, and what I'm going to do is I'm going to do a pair of inserts. So I'm going to insert some data now, and then I'm going to give it about 2 seconds to make sure that our window has elapsed into a new one. And so now I'm going to insert these values for a different sensorID. So now, if we go back to Eclipse, we can see that it took a little while for our program to catch up, but we were able to output two results to the console. So the first one is for sensorID abc, and we can see the start of the window in our value result. So it's, you know, February 7th, 2021, and we can see the exact time whenever the windows started, and then we can see that for the second group of them that came in a little bit later, we can see the at that window started a little bit later. And in both cases, they have that 3, those three values. So that's everything that you need to do to be able to window your data, and you can take this kind of logic, and expand it with custom aggregations or branching or filtering, or what have you.

Summary

Let's recap all of the things that you've learned in this course. First, we covered some of the streaming theory and how Kafka Streams and ksqlDB allow you to expand the events, storage, and transmission built into Kafka by adding a layer of analysis and ETL. Next, we saw how easy it was to get ksqlDB and a Kafka broker up and running, as well as configured, thanks to Docker Compose and Containers. Once the infrastructure was in place, we were able to start writing and deploying queries using a variant of the SQL language. Once we got the basics down with ksqlDB, we came back to theory to talk about how you manage state information, manage time information, and manage the consistency of our results when working with streaming systems. All these things are important whenever you want to start doing streaming ETL or streaming applications. With that out of the way, we rolled up our sleeves to write some Java code and create our first Kafka Streams application. Finally, we looked at going beyond making a single stream processing application and instead looked at how we could do joins and aggregations to look at multiple events and multiple Kafka topics. Now that you've watched this course, you should be well equipped to start using ksqlDB and Kafka Streams in your everyday work.