Applying the Lambda Architecture with Spark, Kafka, and Cassandra

by Ahmad Alkilani

This course introduces how to build robust, scalable, real-time big data systems using a variety of Apache Spark's APIs, including the Streaming, DataFrame, SQL, and DataSources APIs, integrated with Apache Kafka, HDFS and Apache Cassandra.

Course Overview

Hi! My name is Ahmad Alkilani, and welcome to my course, Applying the Lambda Architecture with Spark, Kafka, and Cassandra. We see big data discussed every day whether you're in the field actively working on big data projects, hear about the scale of problems companies like LinkedIn, Facebook, and Twitter have to deal with on a daily basis, or simply listening to the radio about some initiative where big data enabled the analysis and discovery of new insights into the data we have. In this course, our focus will be on building real-time systems that can handle real-time data at scale with robustness and fault-tolerance as first-class citizens using tools like Apache Spark, Kafka, Cassandra, and Hadoop. We'll look at how thoughtful design of your big data applications allows you to combine low latency streaming data in batch workloads. We'll design and build an application from scratch using Apache Spark, Spark DataFrames, and Spark SQL, in addition to Spark's Data Sources API to load, store, and manipulate data. We'll also look at Spark Streaming and Spark-Kafka integration techniques for reliability and speed. We'll also write and Kafka data producer to simulate our real-time data stream feed into our streaming application. And as we dive deeper into the course, we'll look at how you can preserve global state and use memory efficiently with approximate algorithms as we build a stateful Spark Streaming application. And a production application isn't complete without the ability to handle errors and code updates. We'll also learn how to use a scalable NoSQL database and persist your data to Cassandra and HDFS. By the end of this course, you'll feel comfortable building your own fault-tolerant scalable real-time big data systems and act on streaming and batch data with Spark, Kafka, Cassandra, and HDFS as the backbone for the lambda architecture. Before we begin this course, you should be familiar with some programming language, preferably Java, Scala, or C#. But you certainly don't have to be a master in any of these as we'll walk you through a gentle introduction to get you going. I look forward to you joining me in this journey to learn about lambda architectures with the Applying Lambda Architecture with Spark, Kafka, and Cassandra course at Pluralsight.

A Modern Big Data Architecture

Defining the Lambda Architecture

Hi, and welcome to this course. My name is Ahmad Alkilani, and I'll be taking you through an in-depth look at how to build enterprise applications with Apache Spark. In this course, we'll be looking at how to build a modern big data architecture and applying what's known as the lambda architecture. In doing so, we'll look at Spark and Spark Streaming in detail, and we'll have a decent look at how to incorporate Apache Kafka and Cassandra into the mix. In this first module, I'll spend some time describing the problem we'll set out to solve and walk you through what a lambda architecture looks like and how it aims to address these problems. This will give you a good high-level understanding of how all these technologies fit together so it's not overwhelming and you can keep your eye on the prize as we get into the details with each component. I'll also spend some time in this module showing you how to set up your environment and tools we're going to work with. I've put in a lot of effort making sure that you can be up and running with an environment you can use to run our application that simulates a real working environment. In applying the lambda architecture, most of our focus in this course will be around Spark and Spark Streaming. And in this module, I want to make sure you're in good shape to continue the rest of the course. So we'll focus on getting you comfortable using Spark, working with Scala, and getting your environment set up. But, first, we'll start by describing the problem and what a lambda architecture is. We'll start this module by introducing the lambda architecture and describing what a modern big data architecture looks like. While the idea of a lambda architecture isn't novel, the name was coined by Nathan Marz. I like to use a simplified definition for it. It's a data processing architecture and framework designed to address robustness of the scalability and fault-tolerance, be it human or machine fault-tolerance, of big data systems. The keywords here are robust, scalable, and fault-tolerant from both human error and machine failure. These are the key traits a lambda architecture sets out to solve. A lambda architecture attempts to achieve these goals while also balancing between the latency or timeliness of results from data processing and the accuracy of the results. So how does this really happen? A lambda architecture defines three layers. The first is the batch processing layer. The foremost characteristic of this layer is that it holds the master data. So whatever the source of your data, it lands here, untouched, unscathed, in an immutable append-only fashion. This is your record of truth for your entire dataset. This might sound familiar to you, and we've known how to do this and scale it for quite some time now with Hadoop. This is where we look to achieve the most accurate results using any of the processing tools available for a distributed platform like Hadoop. The result of your batch processing results in what we call batch views and are saved using another layer the lambda architecture defines as a serving layer. The serving layer is simply how and where you're presenting this data to consumers, essentially your end datastore. It's been well documented, however, that typical Hadoop processing frameworks are slow, and you might as well grab a cup or coffee while you're waiting for a batch job to finish. The lambda architecture addresses the slowness problem by introducing a speed layer where data streams through. The speed layer's data has another interesting characteristic that further differentiates it from the batch layer in that it's temporal in nature, meaning you can only keep so much information in the stream's known history, which is typically held in memory before you make way for new data. So you get speed but potentially at a price of accuracy. In this course, we'll be using Spark Streaming to represent the streaming layer. Similar to the batch layer, the speed layer presents its view of the world to the serving layer to be joined with the data from the batch layer. The serving layer is where you get a view of both worlds--a latent or old but accurate view of your data as presented by your batch layer, and a fresh low-latency view of the new data as presented by the speed layer. With our choice of technologies, both the batch and speed layers are machine fault-tolerant and can scale horizontally. So what makes this architecture robust and tolerant to human failures? And what do we mean by that anyway? Simply put, it's the fact that we combine the two together in the serving layer that we can make these claims. If, for example, your streaming application exhibits some slowness or you're in the middle of a deployment and you need to bring down some or all of the notes, the serving layer simply ends up with less up-to-date information and will exhibit latency until the batch layer can fill in where the speed layer shows slack. Note that the speed layer could have collapsed completely, perhaps by a bug you introduced unknowingly, hence the human fault-tolerance attribute of the architecture. This could happen all the while your batch layer would continue to provide updates to the serving layer. From an end user's perspective, systems are always online. The only difference in this scenario is the latency at which data arrives. Similarly, the inverse can happen, and your real-time streams will account for more and more of the recent data served until your batch recovers. One final note, the reason it's called a lambda architecture is that the lambda symbol splits into two branches, one representative of the batch layer and the other for the speed layer. So now that we know what a lambda architecture is, let's look at what we're going to build in this course.

What Are We Building?

Let's walk through what we're going to build in this course. We'll start off our application by defining a source of data, so we'll build a simple app that will generate a stream of data that we'll continue to use. The example we're going to use is a generator of click stream data. We'll persist the data in HDFS and perform batch operations against it. We'll also stream the data and produce real-time increments presenting a different set of views of the calculations from the batch layer and speed layer to a persistent store that can be used to serve the combination of the results. And that essentially represents the serving layer in the lambda architecture. What makes this really interesting is that we can use Spark for both batch and real time. Not only can we use the same processing framework, we can also reuse code leveraging business logic and our experience from the implementation of another layer. In this course, we'll be storing our data in Cassandra, and we'll be using Zeppelin to visualize some of our data and also run some Spark and Cassandra commands. Don't worry about the setup. We'll be using a VM image that I've created for you with all the components you need. We'll look at setting up the environment next. In addition to using Cassandra for the serving layer, we'll also use Apache Kafka as our streaming source. We'll go through increments of code changes to our click stream producer to send data to Kafka in a few different formats. And we'll use Spark to sync the data to HDFS and also perform the stream and batch processing. Finally, we'll look at how we can use Spark and Kafka's integration more efficiently and utilize data from HDFS acting as a feedback loop from what Spark needs to pull from Kafka in the event of pipeline failures, shutdowns, or upgrades. In the next clip, we'll get you started with the environment setup. As it requires a reasonable amount of bandwidth to download the VM and tools, I'll use that as an excuse to get you up to speed with Scala. The code we'll be writing in this course is all based on Scala, which Spark and Kafka are actually written with. Don't worry if you've never used Scala before. The following section and as we walk through examples in the course should get you to a good comfort level with the language. But first let's get with environment set up.

Setting up Your Environment: Demo

Let's go ahead and set up the development environment that we're going to use. The first thing we need is the Java Oracle SDK, so let's go ahead and search for that and download Java 8. This will take us to the Oracle website. Go ahead and click Accept for the license. Make sure that you're downloading the development kit. And then select the version that's appropriate for your operating system. And I'll go ahead and click the download for Windows 64 bit, which matches the version of the operating system I have. Once your download completes, go ahead and bring up your installer. This will obviously be different depending on the operating system you have. But go ahead and make sure that you're installing it into a location you're comfortable with. Now on Windows, as the installer progresses, it will ask you for the destination folder for Java. This is basically asking you where your Java home is going to be. So I went ahead and changed my location. Click next for the installer to finish the installation. So I'll assume that you've finished the installation of the Java development kit and the Oracle JDK. And we'll move on to the next step. So next we're going to want to install IntelliJ community edition. So let's search for that. And then notice this is going to be the JetBrains website. Make sure, again, that it's appropriate to our operating system, and go ahead and download the community edition. Go ahead and start the IntelliJ installer after the download completes. IntelliJ is the main IDE I'll be using, so I like to associate it with known file extensions in Windows. And then go ahead and launch IntelliJ. The first time IntelliJ shows up, it will ask you for a default theme. I like using a darker theme, so I'll select Darcula. And then click Next to select default plugins. You'll see IntelliJ lists plugins based on category. We need to install the IntelliJ plugin for Scala, and it shows up as a featured plugin on the next page. If you have an older version of IntelliJ or if this guided UI ever changes and you don't see the same thing, you can still go to Preferences, Plugins, and install the Scala plugin from there. Once installation completes, you should see a screen similar to this that IntelliJ presents you with. So we want to create a new project, and we're going to use Maven as our build tool. So the type of project we want is a Maven project. The first thing we need to do is tell IntelliJ where we have Java installed, point it to the JDK directory where you had the JDK installed, and be sure to select the JDK and not the JRE. IntelliJ will then detect the version of the Java SDK and populate the project SDK drop-down. Click Next, and then we'll fill in the project GroupId as com. Pluralsight and ArtifactId as lambda, and then finally our project name as lambda-Pluralsight. Once IntelliJ comes up, Maven might show you a sign to enable auto-imports. And as I like this enabled, I'll go ahead and select this. Again, this is something you can get from other menus if you don't see this. The view I'm looking at to the left is the project view. If you don't see this, you can click on the small square icon in the bottom left corner and select Project or from the View menu in Tool Windows. Let's go ahead and expand our project, and we're going to go ahead and delete the source directory altogether because we're going to create modules within the main project. Go ahead and right-click on the main project. Select New Module. Again, this is a Maven module, so click Next. Let's give this module a name. IntelliJ now understands that this is a Maven-based module. But we also need to add framework support so IntelliJ understands it's a Scala module. The first time you do this, you probably won't have the Scala library set up. IntelliJ makes this easy by simply clicking on the drop-down menu, selecting download, and choosing the Scala version. We'll be using Scala 2. 11. 8, but anything above 2. 11. 7 will work just fine. Although a quick word of caution--Scala is a little bit picky with major releases, so stick to 2. 11 and don't move on to 2. 12 if you can. And it will take a little bit of time to download the Scala library. So that basically sets up the development environment with the basics that we need. Now let's go ahead and expand this, and you'll see that it only has source, main, Java. So let's go ahead and add a new directory and call that Scala right under main and also mark that directory as a source root. And just to run a quick Scala test, we can go ahead and right-click New and hit Scala Worksheet. This is a nifty feature in IntelliJ, and Eclipse actually has a very similar thing where you get kind of like a Scala REPL but within your IDE. So you can go ahead and add some code here, and you'll see that it automatically starts parsing and executing your code.

Tools We'll Need: Demo

For this course, we're going to be using a rather long list of technologies. And we could certainly spend a few courses just on the administration aspects. So instead of haggling through setup issues, I went ahead and prepared a prepackaged virtual machine image that has everything set up for us, specifically for this course. But before we can use that, we'll need a few bits of software. The first is VirtualBox so we can run virtual machines. If you have VMware, then it should work as well. But if you do face any issues, give VirtualBox a try before you come to conclusions about anything else. It's important to establish a base environment that we're all using. And I'm going to download and use VirtualBox here because it's freely available and pretty simple to use. So, again, choose the version that is appropriate for your operating system. Download it and go through the installation instructions. And one last thing, if you already have VirtualBox installed, make sure you have a recent version of this, so at least 5. 14 or above if possible. The second piece of software we'll need is called Vagrant. So go ahead and point your browser to vagrantup. com. Vagrant is a way to automate the creation of virtual machines and development environments using VMs. It essentially allows you to automate the provisioning settings and standup of VMs. The VM I created for you for this course uses Vagrant to help with this setup. Again, go ahead and download and install Vagrant as appropriate for your operating system. Once we get to the next step, I'll actually assume that you've installed both VirtualBox and Vagrant, and you're ready to go. Now there's one last piece of software that we'll want to install, and that's called Cygwin. Cygwin's basically a tool that provides you a Linux-like shell but on Windows. And we basically want to use this because we're going to be using a lot of commands that will just look a little bit more familiar if you're using the Linux equivalent. Now, again, you'll only want to do this if you're on Windows because Mac already has a decent shell. And if you're on Linux, well, we're only trying to mimic Linux here. Now you can install Cygwin simply by going to Cygwin. com and going through the setup instructions. But I prefer a little bit of a different approach with something that automatically sets me up nicely from the get-go, and that's called Chocolatey. So Chocolatey is simply a package manager very similar to yum or apt-get in Linux environments, but basically a package manager that is targeted for Windows. And what we'll do is we'll use Chocolatey to actually install Cygwin for us. So to install Chocolatey, we actually have to open an administrative command prompt. So I'm going to type Windows S, search for command prompt, right-click, run as administrator, or Ctrl+Shift+Enter will do the same thing. And that should get you an administrator command prompt. Please be sure it's an administrative command prompt or else this won't work and you'll have to follow different instructions. Now on the Chocolatey page, it shows you a command that you'll have to run. So go ahead and copy that and simply paste it here. Hit Enter, and the execution should start. To ensure everything's okay, you can do a choco /?, and that should give you a list of options that you can run. Now, remember, we basically installed Chocolatey to actually get us Cygwin, so let's do choco install cyg-get. Now you can see Chocolatey's actually downloading and installing Cygwin for us. If all goes well, you should see the same prompt. Hit 1 for Yes again, and you're all set. So now that we have Cygwin installed through Chocolatey, we're going to use Chocolatey to actually install two Cygwin packages. And the way to do that is to use cyg-get. So the first package we need is OpenSSH. We need an SSH client. And we'll let that install. And the second package we need is rsync. So we'll do a cyg-get rsync. So if you're not doing this through Chocolatey, for example, and you're simply using Cygwin, then all you need to know is that we need an SSH client and rsync. So in the next segments, we'll actually look at using all of the tools that we've installed and how we bring up our environment. We'll also look at all the components that come up as part of the environment.

Installing the Course VM: Demo

At this point, you should have VirtualBox, Cygwin, and Vagrant installed. And we're going to use Vagrant to pull the VM we're going to use for this course. You're going to need to identify two directories on your computer, one for the VirtualBox VM image itself and another for the Vagrant files, which will also act as the location of where we're going to keep the data that we're going to send to the virtual machine. So that will be our shared folder. The first thing we're going to do to set up the VM is make sure that you have the directory set up. So go to File, Preferences on VirtualBox, and then just make sure under the General section that this points to a location that has enough space. The VM that we're going to download is approximately 8 GB, so that's what you'll need. I'm going to cancel out of this and just keep this open. And then go ahead and open Cygwin. Within Cygwin, we're going to navigate to the directory where I actually want to store the Vagrant box, which is essentially just a file from a Git repository that we're going to clone, and also where the shared directory is going to be where we're going to place data further down in the course. Because Cygwin emulates a Linux environment under Windows, it mounts things a little bit differently than what you would see them on a Windows machine. So Cygwin has all of your drives under cygdrive. And then if I tap Tab, you'll see that I have c/, e/, and f/. And then you'll see this directory, which is where I'm going to store my Git clone and shared Vagrant directory. So I'm going to go ahead and go to f/boxes, case sensitive. And then I'm going to do a Git clone of the Spark Kafka Cassandra GitHub project. So make sure you get this correct. So it's GitHub. com/aalkilani/spark-kafka-cassandra-applying-lambda-architecture. git. So it'll just do a Git clone in this directory. So as you can see, that cloned the Git project. Under the Git project, there's a Vagrant directory, and this is where we're going to initialize our Vagrant box. So back to Cygwin, cd spark, Tab, it'll auto-complete. And then cd vagrant, that's where I want it to be. And then given that you have Vagrant installed, you should be able to simply save vagrant up. And that will use this Vagrant file, which has details of where to get the image from and how to install it. So go ahead and hit Enter on vagrant up, and you should see your box automatically start to download. You'll see it notes that it's importing aalkilani/lambda-Pluralsight, that's the name of the image that we want to import. And, unfortunately, we're going to have to be a little bit patient here. It is an 8-GB image, so it will take some time. You may even wish to move on to the next section of the course as your image downloads. And at some point, you should eventually see it reporting some progress. So I'm currently at 20%. And once it completes the download, you should see that it actually creates a new virtual machine in Vagrant box, and it starts managing it for you. And as it tries to bring up the machine, it'll map a default set of ports that we're going to need. And then it starts booting the machine. At this point, the VM is essentially up and running. You might be tempted to use VirtualBox to start and stop the VM, although it is better to continue to use Vagrant as we did using vagrant up, and I'll show you a couple more commands to manage your virtual machine. So if you want to bring it down, all you need to do is vagrant halt, and that should start shutting down your machine. We're also frequently going to be logging into the VM via SSH. So to do that, you need to run a command called vagrant ssh, and it will log you in under the Vagrant user. So as you can see, it welcomes me to the Linux box that's based on Ubuntu. And the services we're going to be using on the machine are actually all running as docker containers. So a simple docker ps command should show you a list of components that are running on the host. If you ever have a problem with your VM, I'll have a troubleshooting section on my GitHub page that you can take a look at to troubleshoot any problems you might run into. And one last thing about the VM. Remember we said that this would be the location where we're going to store data or have a shared directory. Well, essentially inside the VM, this is mounted under /vagrant. So you can see the Vagrant file here is exactly what shows up inside of the VM. So this in its entirety is a shared directory. So if I were to create a new text file and I ls again, you should also see that text file show up here. Similarly, if you create something on the VM inside of the Vagrant directory, you should also see that show up on the host. And this is how we're going to be sharing files between the host and the VM. Before we move on to the next segment, let's also make sure we can address our virtual machine correctly by adding an entry in our host's file. On a Window's machine, the host's file is typically under C:\\Windows\System32\drivers\etc\hosts. And you'll need to open Notepad as an administrator to edit this file since it's a Windows system file. If you're working on a Mac or Linux machine, the process here is different, and you'll likely need root to edit your etc\hosts file. So all we need here is to map lambda-Pluralsight to the local loopback IP address defined as 127. 0. 0. 1 and save our file. So now if you open a web browser from your host machine, you can reference the VM as lambda-Pluralsight instead of just localhost. This isn't too important for about applications. However, it is really important for us to be able to reference the Hadoop cluster correctly. So this needs to be set up.

Fast Track to Scala: Basics

Scala is a language that runs on the JVM very similar to Java but has functional, as well as object-oriented roots. Scala also defers from Java in that it's a lot less verbose, and the language and compiler provide a lot of niceties to reduce clutter and the amount of code you end up writing. No matter the programming background you come from, all I ask is that you come with an open mind when looking at Scala. It may look a bit weird at first, but I promise you will feel at home by the time we're done with this course. If you're already comfortable with Scala, feel free to quickly skim through this section, albeit there still might be something for you to learn here as well. Let's start with the basics--variable declaration in Scala. In Scala, you declare a variable by prefixing it with either val or var. A val declares a variable that cannot be changed and is the preferred style. This is similar to a final in Java. In our little example here, I declare a val text of type String. Notice also that the type of the variable follows its name opposite of Java and many other languages but for good reason. Scala has a lot of type inference, so in many cases, we don't end up explicitly calling out the type of each variable, although it can still be helpful to do so when it's not clear what the type is from the context for readability. In any case, the val text of type String is assigned a string literal value, not another word count example. Don't worry, there'll be no word count examples in this course. If you try to reassign another value to a variable declared with a val, you'll get a compile error or the IDE will yell at you. On the other hand, declaring a variable with var makes it a mutable variable. There's a good reason why we like using val instead of var wherever possible because it eliminates sharing of objects as a result, and your code is more readily parallelizable. So variable definition is down. Let's move on. Remember the type inference I was referring to earlier? Let's use it here. So in this case, I could've just left the type out, and Scala based on the assignment will determine the type. A lot less typing and still clear what's going on. I could also argue some clutter also went away, so it's even clearer what the code is doing without any obvious types getting in our way. Oh, and by the way, we're going to be using IntelliJ. The shortcut to view Scala types, if you're ever wondering what the type of a variable is, is to use the Alt key and equal sign. Moving on, function definitions in Scala start with def, then the function name, and again notice here the parameter, then the type of the parameter, and then just as with parameters, return types for functions follow the name and definition. So we're saying this function returns a string as this function also returns something, so it also has an equal sign. Every line in Scala is an expression. So essentially we're saying, Here's a function definition. And it equals the expression defined in the code within the two curly braces. Let's look at the body of the function. There are a few things going on here. First, notice the s in front of the string. This is called string interpolation in Scala. String interpolation allows users to embed variable references directly in processed string literals. But let's not make this more complicated than it is. So think of it as a simple shortcut to a string format in Java or printf formatting in most other languages but in a much more concise syntax. Anything prefixed with a dollar sign inside of the string is assumed to be a variable you're placing in that location. Notice here that there is no return keyword. Every statement in Scala is an expression and returns itself. So while you could type return if you really wanted to, and Scala will accept it, most code checkers and the IDE will flag this as non-idiomatic Scala. Just keep in mind that the last statement is the return statement, and you're good to go. Let's have another look at type inference. So, again, Scala here can infer the return type of our function. So in reality, I won't have to provide it, and the code shrinks even further yet still very readable. One feature in Scala that throws off many is that functions can have multiple parameter lists. You know that thing a function takes in parentheses that defines the parameters a function accepts, well Scala lets you have multiple of those. And they are referred to as parameter lists. So Scala allows for multiple parameter lists. Multiple parameter lists in Scala are very useful when the last parameter is an implicit parameter. I'll introduce that later, but here's a simple example, albeit not very useful. We'll look at a better example next. So here I have a function sayHello that has multiple parameter lists. The first list only takes one parameter called name. The second list takes a parameter called whoAreYou. The interesting thing is that this parameter's type is a function that takes no input and returns a string. Scala is a functional language, so functions are first-class citizens, and you can pass them around just like that. Let's look at how you can call this function and, better yet, utilize the multiple parameter lists sayHello provides. So let's define a function that matches the requirements for the whoAreYou parameter that sayHello expects. We'll create a function called provideName that takes no parameters and just returns the string literal Scala. Then we'll define a val to store the result of calling the sayHello function, and we'll make that call, so sayHello of test, and we'll pass along provideName for the parameter for the second parameter list. Now that wasn't too useful, was it? Let's make use of Scala's anonymous functions or lambda expressions, not to be confused with lambda architecture. So, again, we create a val called faster, and we call sayHello passing test. Instead of providing a predefined function that matches the type of the required parameter whoAreYou, we'll create an anonymous function inline, and the beauty of the multiple parameter lists syntax lets us write code like this. The second parameter was provided as an anonymous function that matches its signature. The function is anonymous because it has no name we know of, and we'll look at plenty more examples like this. So don't worry if this is still too much to handle at this moment.

Fast Track to Scala: Language Features

Scala has an advanced feature called implicits. Again, not to complicate things, if I were to tell you I'm 5 feet 10 inches or 178 cm, wouldn't you implicitly assume I was talking about my height? Implicits in Scala are kind of the same. Let's look at the example. So we have the same old sayHello function we've been using. But now in the second parameter list, it expects an implicit parameter of type string. If we defined a val and prefix it with implicit, the compiler will implicitly send that to the function as a parameter myself because it matches the expected type. The variable's defined as an implicit, so I'm telling the compiler it's okay to use this to sub implicit asks. And my implicit variable, myString, is in scope. The compiler obviously won't grab implicits that match type from any scope. It has to be within the scope of what you're calling. So the call to sayHello in this example returns Hello (test)! My name is (implicits). Let's look at another example. This is very similar to the example where the second parameter's type expected a function that returned a string. But the only difference here is that I have a function that matches defined as implicit. And sayHello also notes that it can take an implicit. So all is well, and calling sayHello returns what you'd expect. There are only two things that might look a bit interesting here. First, classes in Scala take parameter lists. The parameters define the class members and, based on whether you define them as public or private, will have getters and setters already defined for you. So class fastTrack has two member variables, name and myself. The default is that they will be declared as public. The second thing going on here is that you don't see a primary constructor. The body of the class is the constructor. So the constructor basically creates a val, greeting, and calls the sayHello function on the class member variables, name and myself. Let's take a second look at class parameters that define member variables. We can prefix these with val or var, and that essentially controls if they get getters and setters or getters only. Similarly, you can prefix with private to indicate a private member variable. We'll discuss a special kind of class called case classes next. Case classes are a special kind of class. They're not a programming construct within the language as much as they are a syntactic sugar that Scala gives to its programmers. The short of it is that if you prefix a class with the case keyword, it will have Scala add a few freebies to the class. It essentially makes the class serializable, adds a toString method, adds a hashCode implementation and an equality implementation, among a few other things. It also adds an auxiliary constructor such that it makes case classes not require the new keyword to create an instance. So if you see code that looks a little uncanny because you're getting an instance of a class without using the new keyword, it's usually because it's defined as a case class. You could obviously implement all of this yourself if you wanted to. Case classes have all parameters prefixed with val by default. And they support something called pattern matching in Scala, which we'll look at next. Now that you know how case classes work, we can venture into pattern matching in Scala. Pattern matching isn't tied to case classes necessarily, but it's used frequently with case classes. Pattern matching is used quite a bit in Spark code, and so it's important to understand. In this example, I have an abstract class person. Notice the string interpolation in the fullName method. We also have an implementation of the abstract class called Student. The Student class is defined as a case class. We define a val, me, with first and last name, Ahmad Alkilani, and Id 23. We then define a function, getFullID, which takes a parameter called something of type T. T is a generic or template as known in various languages. Java uses angle brackets to define template variables. In Scala, it's square brackets. Also note that in Scala, we're saying that this template variable must be of type Person. And with the less than sign that fronts the typical colon for the data type, we're defining a bounded template variable. The less sign means that the provided T must be a person or a derivative of a person. This is an upper type bound on T, which declares that the type variable T refers to a subtype of person in our example, a nice way to guarantee types within your implementation. In fact, you'll get a compile error if you pass anything that doesn't match. We digress from the subject of pattern matching a little bit, but it was important to explain what that meant. Again, you'll see this in Scala and Spark code. Now to the guts of the getFullID function. We're applying a pattern match similar to a switch statement in most languages. However, pattern matching is much more powerful because we can match on types as well. In our example, we take the variable something and perform a match against it. The first case tries to match it to a student. And in the case it matches, we also do parameter extraction where we extract the values $fname, $lname, and $id from Student. This is possible because Student is declared as a case class, so case classes also implement pattern extractors. Instead of extracting a val of type Student and then in the expression that follows use the val to grab the class variables $fname, $lname, and $id, we simply extract them as part of the pattern match and use them in the expression that follows. So in case the type of Person is a Student, then the full Id is defined as $fname-$lname-$id. Our match expression doesn't end here, however. If the type is not a Student, we define the full Id as the full name, which simply calls the fullName function defined on the Person abstract class. If you're wondering about the return statement, remember the last expression is the return statement. This match statement is an expression. In fact, it's called a match expression as everything else in Scala is an expression. So the result of the match statement is returned, and because it's the last expression in our function definition here, then the result is what the function returns. The match expression essentially is our return statement. This example was made somewhat complex on purpose. First, it shows a real example of code you'll encounter when working with Scala. And I didn't want you to feel uneasy reading code like this so you'll always have a reference example to go back to. I also want to make sure we cover real-world examples in the code we tackle further along in the course instead of cutting the course short because we haven't covered some examples like this. We're just about done with Scala so hang in there. Next up, implicits. Implicit conversions in Scala are similar to the implicit parameters we've seen earlier. Something happens implicitly. With implicit conversions, you can implicitly convert a variable from one type to another. If you're familiar with the extension methods in other languages, this is how Scala achieves that concept. It allows you to implicitly convert a type T to a type R where R defines additional functionality that is made available to type T as long as the implicit conversion for type R is in scope. That sounded complicated, so let's simplify it. The example we have here is that we have a string literal so a type String. And for some reason, we were able to call the scalaWordCount method. How is this possible? Well our class stringUtils is declared as an implicit class that takes a single parameter of type String and defines the method scalaWordCount. Because this class is an implicit class, it's telling the compiler, I can implicitly take any variable of type String and convert it to an instance of myself. By doing so, the string remains intact but also get the scalaWordCount as an extension to the functionality. So when the implicit class stringUtils is in scope, the compiler happily takes the string Spark collections mimic Scala collections and makes available the scalaWordCount method. Because implicit conversions seem a bit magical in nature, most IDEs like IntelliJ understand that this is happening and will literally underscore that fact so you're paying attention to what's going on in case there was something unintentional there. So why are we looking at implicit conversions in Scala anyway? Well because Spark uses them all the time. It's how Spark lets you automatically convert from one type of RDD to another. More on that when we get there. One little tidbit--Scala's programmers like to think they're pretty cool, so they like to call implicit conversions like this as the Pimp My Class pattern. If you hear that, now you know what it means. This example also introduces the Scala collections library. I know I promised no word count, but I couldn't resist. The methods start off by creating an array of type String by splitting on whitespace, so we end up with a word per element in the array. Then we use a Scala collections API groupBy method on the array of String. This creates a map of each value and a list of the values. The type of the grouped variable is Map of String, and Array of String. All that's left is to count the records in the array for each group. So we use a function called mapValues. Mapping over a collection in Scala essentially lets you apply a function for every record in the collection and returns the result of that function as the new element in the collection replacing the existing element. In this case, we're using a special kind of map called mapValues, which assumes the input type is a type that has a key and a value, so we're only mapping on the values for each key. So in this example, we're only mapping on the array of String part of the original map. So, again, for this example, if we had a word Hello and an array of all the occurrences of Hello, then mapValues returns the key as is and performs the map operation against the values, in this case the array of occurrences of the word Hello. The map operation here is defined as a function that takes a value. In this case, the value is the array String and returns the length of size of the array. The countPerKey variable now has a map of type String and Long, so a map of words and their counts. It's noteworthy that Spark's collection API is very similar to Scala's collection API. Also worth noting is that mapValues is called a higher order function in Scala. Remember how functions are first-class citizens in Scala? Well a higher order function is just a formal definition of that. It's a function that takes another function as one of its parameters or returns another function or does both. In this case, mapValues is an example of a function that takes another function as a parameter. So mapValues essentially asks for a function that you provide where it will use to iterate over the collection and apply the value of each element to the function provided returning the result.

Fast Track to Scala: Collections

Let's take a closer look at Scala collections. So here we have a val, myList, defined as a list of Strings. We then use Scala's map function to iterate over each element in the list providing a function to map of what to do. The function we provide here is an anonymous function or a lambda function. All it does is take each element, and we're giving each element the val name s. And it applies an s. toUpperCast to the element. One by one, this lambda gets applied to the elements and the result returned in a variable called Mapped. Notice here Scala's type inference in play. We could have defined this as s of type String goes to s. toUpperCase. The arrow here in the syntax is used to define an anonymous function and reads goes to. As you can see, the return result is still a list of String with each string capitalized. Another extremely useful method from the Scala collections API is flatMap. It's similar to map in that it is a higher order function meaning it takes a function in this case that is used to apply to each element in the collection. What flatMap does differently than map is that it unboxes the returned results if you will. So it has the property of potentially exploding the number of records returned or reducing the number of records returned or leaving the count the same. Let's look at this example and explain how flatMap works. For each element s in my list, a lambda function is defined that operates on s. So far nothing different from what map does. The lambda function here defines a list of filters. It then checks to see if the provided element, s, is one of the items we're looking for. And if it is, it returns a None. If the if statement, on the other hand, returns false, it returns a Some of s. Now None and Some of any type T are derived classes from another class called Option of type T. So you can return a None, which essentially is a special instance of an Option of type T, that says I have nothing. Or you can return a Some of type T. In this case, we're returning Some of S, so Some of String. An Option is an invaluable part of Scala which lets you control your code easier and in many cases avoid using nulls and checking for nulls because an Option of T has two possible implementations--either None or Some of that type. So we can essentially cast the return type and do pattern matching. Now that we know that None and Some are subtypes of Option of T, we now know that the return type of this if statement is an Option of type String. An Option acts as a wrapper for its value. And because we said flatMap unboxes a value, what we get in the case of Some of s as a result of flatMap is simply the value s. flatMap unboxes the Some wrapper. But what happens with None? So flatMap unboxes None, has no value left, so the end result is that the element is filtered out. So you'll see flatMap used this way again and again to substitute a map then filter operation. Some people like to think of it as a map and filter, but filtering is really an intrinsic property of the fact that it unboxes whatever you give it back. And because we hand it an option, it will end up filtering those values where the return type is None. I won't spend more time on Scala and its collection library because we are going to see many more examples as we work through spark code. If you need a little bit more help with Scala collections, especially map and flatMap because they are used extensively in Spark, you can check out another course I have on Pluralsight that walks through a better example step by step. The course is called Data Science & Hadoop Workflows at Scale with Scalding. See module 1, Scala Map, and module 2 that talks about flatMap for a better look at each. Congratulations. You're now a Scala programmer.

Spark with Zeppelin: Demo

Now that we have our VM up and running, go ahead and point your browser to localhost port 8988, and it should bring up the Welcome page for Apache Zeppelin as you see here. Zeppelin is a web-based notebook similar to the IPython Notebook if you're familiar with that. For this course, we're going to be using Zeppelin as an interactive UI for Spark and some Cassandra work. Feel free to check out the Apache Zeppelin web page. Zeppelin itself comes with a lot of interpreters. The ones we're most interested in here are the Spark interpreter and the Cassandra interpreter. So let's go ahead and create a new note, and we'll give this the name Spark Intro. And as you see, we have a new notebook with the section of the window ready for some input. On the right-hand side, you'll notice a small settings cog. If you click on that, you'll notice the binding for the default interpreter, so the default interpreter that binds to this notebook is Spark and then markdown, Angular, shell hive, etc. So as long as you're using some of the bindings on this list, you should be fine. And the bindings in that window also define the order in which Zeppelin will work through them. So let's apply our Scala skills and work out a small Spark example. by default, Scala and Spark already bind, so we can go ahead and try and create a case class. We'll call the case class Item, and it'll have two fields--itemName, which is a string, and itemNumber as an int. And then we're going to create a range. In Scala, a range is very easily created with a syntax like this. And now that I have a range, which essentially represents a collection in Scala, remember the collections API I can actually map over this, so now I have an anonymous function here. So for each element in the range i, I'm going to apply this function. And the function really is that we're going to create a new instance of the Item case class. And, again, because it's a case class, I don't need the new keyword. And also we'll use another technique that we learned, which is string interpolation, and we'll just say name, and then of i, and then just the value i as the item number. And let's assign this to an items variable. Now I know we haven't looked at any Spark code yet, and we still don't know any details about Spark. But to make this interesting, we're going to write a very simple Spark program. In Spark, everything starts with the Spark context. Within a Zeppelin Notebook, a Spark context is already created for you called sc. So there's a val called sc that's already an instance of an object of a Spark context. Also in Spark, there are multiple ways how you can create an RDD. The RDD is the main abstraction that Spark uses, and it stands for resilient distributed dataset. You can create an RDD by loading data from a file or distributed file system or by simply loading data from a collection in memory. So let's go ahead and try that. So we'll use this Spark context variable. And we'll use the parallelize function on the Spark context, and we'll provide it the items collection. So what Spark is doing here is it's taking the items collection, and then it's spreading it across the Spark cluster. Remember, we're running against a VM here, so the Spark cluster really is just one machine. Now let's go ahead and assign this to an itemsRDD val. And then just to show you some of the capabilities of Spark and Zeppelin, we're going to go ahead and create a DataFrame from this RDD. Now Spark provides some implicit conversions (if you remember when we talked about implicits and implicit conversions) that allow you to convert an RDD to a DataFrame. And it can do this without any additional input from you because we've actually provided it with a case class. And a case class provides extractors so Spark actually knows what to call the column names and the data types of the column names. Now I'm going to go ahead and run this. You can run it in two different ways, so you can either click the triangle here in the top right corner or Shift+Enter will do, and it'll give you another extension to the sheet. So as you can see, it actually defined the class item. It also defined the items collection and the itemsRDD and itemsDF. The first time you run this, the Apache Zeppelin Notebook will actually have to initialize the Spark context. So that will take some time. Don't give up on it. It takes about a minute or two. So now I want to go ahead and use our itemsDF and actually register a temp table in Spark and call this items. So what this does is it actually uses the information from the DataFrame because it knows the data types of the columns and the column names. So you can actually register a temp table and then use that temp table as if you're just writing SQL against it. However, to tell Zeppelin that we're writing SQL now, we need to change the binding. So we'll select SQL as the binding, and then we can simply select \* from items. And then Shift+Enter. You'll see it actually grabbed the items in the list. And Zeppelin also comes with some very simple useful visualizations that you can select. You can also view the settings. It understands that you have an itemNumber field, and itemName. Again, this is by defining the temp table. And everything eventually came from the case class. So this was our very first Spark example. And we're going to go through a lot more Spark examples down the road. I just wanted to take this opportunity to showcase the Apache Zeppelin Notebook that came up with our virtual machine.

Summary

At this point, we barely scratched the surface with what we're going to learn in this course about the lambda architecture. However, by now you should have a generic understanding of what a lambda architecture is, how the batch and speed layers come together in the serving layer, and how that helps you achieve fault-tolerance. We've also briefly discussed Spark and why it's actually a very good fit for a lambda architecture, specifically because we can share logic between the batch and speed layers so you're not working within two different contexts of two different implementations or two different systems to achieve the same goal. And in preparation for all of the work that we're going to do with Spark, we went through a quick fast track to Scala. In the section on Scala, we discussed functions with parameter lists, how they can be useful with higher order functions that take other functions as parameter. We also saw an example of that with map and flatMap when we discussed Scala collections. We also looked at parameter extraction and part of what makes case classes such a breeze to use. We also looked at string interpolation, pattern matching, in addition to looking at a few other Scala shortcuts like type inference. And also by now, hopefully you have your environment set up. So we've set up IntelliJ, Cygwin, Vagrant, and VirtualBox. And we also downloaded the VM that we're going to use in this course. And once our VM was up, we took a quick look at Apache Zeppelin and built a small Spark example. Within that example, we looked at how the Spark context is really the center of the universe for Spark and how we can create RDDs from different sources, whether an in-memory source or reading data from a file, which we'll actually look at next when we discuss implementing the batch layer in the next module. And we also looked at a very simple example of the use of DataFrames. In the next module, we'll run a very quick intro to Spark and then we'll roll up our sleeves and start working on our batch layer. This has been the first module in the course, Applying the Lambda Architecture with Spark, Kafka, and Cassandra.

Batch Layer with Apache Spark

Introduction to Spark

Hi! This is Ahmad Alkilani from Pluralsight. In this module, we'll start building the first incremental step in a lambda architecture using Spark. We'll build the fundamental pieces of the batch layer. I'll introduce the types of queries we'll be running with Spark and a few different techniques to get the job done. In this module, we'll also demonstrate Spark's RDD API, DataFrames API, and Dataset API, in addition to caching in Spark, as we perform aggregations and build the first phase of our batch layer. We'll start off this module with a very quick introduction to Spark and why it's a good fit for our lambda architecture. We'll also quickly cover Spark's components and how they interact. I believe it's critical to understand at the very least some of the major components and how they affect the execution of your program. So let's get started. Spark was originally developed in 2009 at UC Berkeley's AMPLab and was open sourced in 2010 as an Apache project. Perhaps one of the most interesting facts about Spark is that when Matei Zaharia first conceived it, it was intended to demonstrate the capabilities of Apache Mesos, which also came out of AMPLab as an open source resource scheduler. Spark was intended to show how you could implement a framework for Mesos that ran distributed applications aimed at data processing. Spark then grew into its own project. The good news is that Spark started out as a distributed computing platform so this was never an afterthought. Spark continued to evolve, and Spark 1. 0 brought us Spark SQL and soon thereafter the Data Sources API. Spark 1. 3 introduced the DataFrame API, which brought some structure to data. And Spark 1. 6 brings us a Dataset API, which adds to the DataFrame API by adding type safety to the structured table representation of data that DataFrames bring. Spark is now in 2. 0. 1, and there's a big focus on streaming and specifically something called structured streaming. This continues to build on the frameworks that API Spark brings to the table. However, as of Spark 2. 0, structured streaming is still in very early beta or alpha stages, so we won't be focusing on that just yet, and we'll continue to work with Spark Streaming, Spark SQL, and the Data Sources and DataFrames APIs. So Spark is a general engine for large-scale data processing. It is very similar to MapReduce in that it scales horizontally and can operate on a massive amount of data. It has fault-tolerant characteristics and basically everything that you now come to expect from a distributed system. Sounds familiar? That's what MapReduce promised in my opinion but never delivered be it in the speed or ease of programming. The MapReduce API was bloated, not very productive, and lacked abstractions to provide opportunities for optimization, which led to the rise of higher level frameworks like Cascading, Scalding, Pig, and Hive. The evolution of MapReduce in core Apache Hadoop has been the introduction of Apache TEZ. With YARN now being a generic resource manager and scheduler, other frameworks can run on Hadoop now besides MapReduce. TEZ is one of those frameworks that was built to address some of the inefficiencies with MapReduce. TEZ performs specific optimizations by building its own directed acyclic graph or DAG based on your program and optimizes that DAG with a substantially less amount of data hitting disk and passed on through memory instead. Spark in a very similar fashion built its own execution DAG as well and has its own optimizations and scheduling for executing that DAG. The core strength of Spark's performance when compared to other frameworks is that it can utilize memory and cache objects efficiently and that it also keeps a lineage graph of your operations so it can re-compute on failures. These are two of the fundamental things that the resilient distributed dataset implementation in Spark is all about. Spark's strength is also the programming model and the total cost of ownership. Apart from being a higher level abstraction, Spark offers multiple building blocks so you don't have to constantly switch between different frameworks if you're doing streaming versus batch of SQL. In other words, the speed and efficiency of developing applications on Spark is far and beyond what you would get with MapReduce or even with higher level abstractions. Spark's core provides a set of capabilities for scheduling and execution. Spark SQL brings a SQL layer to Spark so you can run SQL queries against distributed Datasets. Spark Streaming implements a streaming API by extending the batch framework running smaller micro-batches. Spark also brings MLLib, which is a machine learning library, GraphX for graph computations, and Spark R. As Spark continued to evolve, a DataFrames and Dataset abstraction were also reduced. Don't let the names scare you. Think of a DataFrame as a table representation of data. That's all it really is. With a table definition in any database, the database knows the data types of columns and names of columns, and that's precisely what Spark takes advantage of. Knowing the types of columns opens up the opportunity for optimizations, compression, and column storage techniques, in addition to allowing for an additional LINQ-like DSL to query data. A Dataset in Spark is just a DataFrame on steroids. It adds type safety further protecting the programmer, you, from expensive runtime exceptions. It also gives Spark more insight into what your functions are doing against the data while giving the flexibility of executing at a level similar to RDDs so you get the best of both worlds with RDD-like APIs but with the added performance boost you get by executing against DataFrames. Again, we'll see this in action in our code, so don't worry about all of this registering just yet. Finally, Spark's popularity is also hugely related to the fact that it supports a good list of programming languages and execution clusters. While Spark was originally built in Scala to execute on top of Mesos, it also supports Python, Java, and R, in addition to coming with its own scheduler, if you don't have Mesos, and the excellent integration with YARN, which we see a lot of innovation for new Spark features happen first. A good example for that is Spark's dynamic allocation, which came out first on YARN before any other scheduler. The fundamental abstraction for Spark is called an RDD, which stands for resilient distributed dataset. An RDD is a distributed collection of objects or data represented to the user as a familiar collection in any programming construct. So in this example, we have a Scala collection of type Array of some type visually represented here by a block, and this array has a few items. We also have a Spark RDD that looks and feels like a normal Scala collection with the difference being that it operates on elements stored on different machines. Yet you as the programmer still see this as a normal Scala collection. This means you can create what appears to be a simple list of items, and you can operate against the list almost as you normally would with any other Scala collection, albeit with a super set of added functionality specific to the distributed nature of it actually being a distributed collection. Again, in this example, we map against both the Scala collection and the Spark RDD in exactly the same way. The difference is in how that operation takes effect and where the data is stored. The resilient part of the RDD definition lies in the fact that Spark keeps track of data lineage as we discussed earlier. We'll start looking at some Spark code shortly, but what I want you to take away from this is that Spark generates its own execution DAG, can perform optimizations on it, cache data on the way, and it's resilient in nature, and, finally, that there are components that build on top of the Spark core like Streaming and the Spark SQL APIs. In the next clip, we'll quickly take a look at Spark's execution components before we start writing some code in our first program.

Spark Components and Scheduling

It's important that we start on solid ground with our understanding of Spark components, and that includes scheduling as well. This not only helps you become a better programmer but also helps you understand parts of the ecosystem. And speaking of Spark's ecosystem, let's start with Spark's relationship to Hadoop. There are dozens of examples that compare Spark to Hadoop and especially when it comes to performance. However, there's more to be said about that story. Spark's relationship to Hadoop starts with the fact that it can read and write to HDFS, so Spark natively integrates with Hadoop APIs and can write in a variety of file formats that are native to HDFS. Spark also carries tight integration with Hive and Hive's metastore so it can read and write from tables that are defined and stored in Hive's metastore. In addition, Spark SQL also carries a special SQL context called a HiveContext, which actually uses the Hive query languages. And as we know, Spark needs a cluster manager to run on. And while Spark has its own cluster manager, there are definite advantages for running Spark on YARN or Spark on Mesos, especially if you're running HDFS on the same cluster and getting data locality. So Spark uses YARN as a cluster manager to schedule the execution of Spark Executors. So in reality when comparing Spark to Hadoop, what people are really comparing is Spark to MapReduce. And from YARN's point of view, they're both different applications that need to be scheduled and require resources. So what does a Spark application or program really compose from? A Spark application consists of a driver program. That's essentially your code that defines what actions and transformations are done against the data and which data sources to consume from and which data sources to emit to. That's essentially your driver program, which is driving all of the execution. Your driver program is also responsible for bringing up the all-important SparkContext, in addition to understanding how to communicate with a cluster manager be it YARN Mesos or stand-alone, and by that we mean to request resources, and those resources, which are represented here in the form of worker nodes, contain an executor. An executor is simply a JVM, which can spawn off different tasks, and these tasks perform specific actions based on what the driver tells them to do. So essentially the driver is acting as a scheduler in and of itself. And then each executor also carries a portion of its memory that it keeps as cache, and that's essentially what the Spark cache is. It's a portion of the JVM memory inside of the executor. Now that story is changing a little bit where Spark can now actually store data off heap, so outside of the JVM. But let's not complicate things here. The essential thing is to understand that the driver requests resources from a scheduler. That scheduler in our case is YARN. And then once the program gets the resources it needs in the form of executors, it's the job of the driver to then schedule the execution of tasks that it sends to those executors. So if you look at this a little bit differently, you probably already realize that there are actually two schedulers involved here. The first is the cluster manager scheduler, and the second is the Spark scheduler itself. The cluster manager is responsible for responding to resource requests for executors, and the Spark scheduler is responsible for translating your transformations and actions into a DAG that can actually be executed on the executors and then sending that and scheduling that for execution on those executors. Now there remains the question of where the driver program actually runs. And Spark actually differentiates that between two different modes--client mode and cluster mode--and their specific action of what happens when you're running this on a YARN cluster. When the driver or Spark scheduler is submitted as client mode, then the machine or client host that you're submitting your application from actually hosts the driver code. And from there requests are made to YARN to submit the application and request for resources, and then YARN creates an application master and containers that then communicate back to the application master, which then the driver becomes aware of. The very important distinction here is that your host is now responsible for the livelihood of the driver. So if your driver goes down, then it's very likely your entire application goes down unless you have some means of recovery. The second approach to scheduling an application is to request for cluster mode, and that means that the driver itself, essentially your Spark scheduler, is introduced as a container within your cluster manager. So in YARN's case, the driver actually lives within the bounds of the application master. And then it communicates with executors within other containers just as it does in the client mode. The distinction here is that your driver is part of the application master, which is just a special type of container in YARN, and that means that your cluster manager actually manages its livelihood. So it's no longer the job of the client or the host submitting the application to remain alive for your application to remain in the healthy state. It's now the job and the responsibility of the cluster manager to do that. And just by having that, you also get better availability guarantees. And there's one more very important distinction between client and cluster mode apart from who owns the driver. When you submit an application in client mode, because your host actually still has and owns the driver, then you can communicate the driver and continuously send different instructions for your driver to execute. So if you decided you wanted to perform additional action or transformation, then you can feed that into the driver, which then translates that to a DAG and then executes that against Spark's executors. And a perfect example of using that mode effectively is the Spark REPL where you get to launch an application even though it's running on a cluster, but because your driver is local, you can continuously feed it different types of instructions. And the inverse of that is also true. If your application is submitted in cluster mode, then you no longer have interactivity with your driver. But, again, in that mode, you gain different advantages like better fault-tolerance and restartability. Now in YARN inside of each one of these containers is the Spark Executor, which carries different tasks and the cache. And the way Spark's splits out its body of work that it has to do is that it creates partitions that it needs to work on. And every partition consumes one of these task slots if you will. If you're familiar with MapReduce, this is very similar to input splits where an input split consumes a map task. So depending on the way that Spark partitions your data, each one of those partitions will consume one of these tasks. We'll end this segment with an understanding of what Spark actions and transformations are. As we know, Spark bases everything on RDDs, and any manipulation of the RDD, be it transforming the data itself or perhaps joining multiple RDDs, is called a transformation. And as long as you're performing transformations, you're simply adding more steps to the DAG that the Spark driver's accumulating for you. Now once you perform an action on those transformations, be it you're actually performing a count, storing the results in a database, anything that essentially requires the transformation steps in your DAG to be realized, that is what we call an action in Spark. Your DAG itself is composed of multiple stages, and then when you perform an action, that spawns off a job that gets sent to the executors for execution. And every action on each RDD will repeat every transformation that you've laid out with one very important exception and that's in the case that you've actually cached data as part of one of your transformations. So that means subsequent actions won't have to start from the very beginning and can start from where you specify data to be cached. So the fact that you can choose to cache and where to cache is what makes Spark very good at performing iterative algorithms. In the next segment, we'll put all this information to practice and start working on our first real Spark program.

Getting Started: Log Producer Demo

In the segments that remain for this module, we're going to be spending most of our time in the IDE working with Spark code. However, before we can start working on our Spark program, we need a dataset to work with. In this segment, we'll build a small LogProducer program that simulates the production of clickstream logs, and we'll start off by looking at an example of what those logs look like. Then we'll dive into the code, and I'll introduce a configuration library called Typesafe Config that we'll use to configure the behavior of our LogProducer. This is a good opportunity to display what goes into building an enterprise application. Typesafe Config uses a superset of JSON notation called HOCON, which stands for human optimized config object notation. The Typesafe Config library and HOCON allow for the developer to provide layers of overrides for properties to facilitate easier deployments and configuration changes. Finally, before we get started, I'll note the lifecycle of iterations our LogProducer will go through throughout the remainder of the course. In this segment, we'll create the essentials of the LogProducer and have it produce to a single output file and use that as input in our Spark program. In the next module, when we discuss Spark Streaming, we'll modify our code slightly to produce to multiple files simulating a stream of data. And then as we progress with the course in module 4 and discuss Kafka, we'll modify the code yet again and write a Kafka producer this time truly sending a stream of data to the Kafka broker we have set up in our VM. And we'll continue to update the code to produce data in Avro format, and we'll see how to operate on that in Spark. So what does the format of the dataset we're going to produce look like? The file we'll produce is fairly simple. It starts with a timestamp that we occasionally adjust and increment to simulate change in time. We then have a set of fields representing a referrer, an action the visitor to our website has taken. You'll see most of these are page\_views, and you'll have an occasional add\_to\_cart and purchase events. We'll also create a random list of visitors, pages including a tracking of the previous page if the referrer was internal to our website, meaning the action was a result of being on a previous page on our website. This data is random, so in this example, I have no previous page to display. And we'll also show the current page where the action took place and a product related to that action and page. The list of referrers and products we'll use will come from two files that we'll load and use. The products file is just a random list of product names I found online, and the referrers file is a simple list of major referrers you would expect. If your subscription allows you access to the source code, then these sample files are included, but you can just as easily create your own mockup. There's nothing special about these. So let's get started with our code. Let's go ahead and start from where we left off in module 1. The first thing I'm going to do is remove this module and then follow that by a delete. And then we'll start fresh with a new module. We want our new module to be a Maven module, so go ahead and click Next. And let's go ahead and give our artifact a name, spark-lambda. Click Next. Also spark-lambda here, and then let's make sure that that's also the directory that's used. Also in case you see this pop-up, I always like to enable auto-imports from Maven projects. Now that we have our new module created, let's go ahead and add the framework support for Scala. So if you expand this now, you'll see source, main, java. We need a Scala directory, so we'll go ahead and create that. And I know we're not going to be writing any Java, so we'll delete this and mark the Scala directory as a source root. Also make sure your resources directory has this little icon. If it doesn't, just right-click on it, and then select Mark as Resources Root. So I'm going to go ahead and grab the products and referrers files as resources. So you'll see this is just a random list of products for the products, and referrers has Google, Bing, Yahoo, Twitter, etc. I'll also close this POM file. Go ahead and open the main project POM file, so not the POM file for the module, but the main project POM file. If you still see module 1 here, go ahead and delete it. And what we're going to want to do is add the dependency for the Typesafe Config library. So I'll create a dependency section here, and then I'll copy and paste the dependency for Typesafe Config, so com. typesafe and config, versions 1. 3. And now just to keep things tidy, I'm going to create a package for our LogProducer called clickstream and also a package for our configuration called config. Now Typesafe Config expects to see an application. conf or resource. conf file in the resources as part of your jar or available to it on the Java class path. So we'll go ahead and create an application. conf file. And I'm just going to copy and paste a set of configurations here. So these files are actually hierarchical. That's how the HOCON notation actually works. So clickstream could have another section under it called logs for example. So what we're seeing here is that we have a clickstream section, and we have a couple of configuration parameters. We're defining the number of records, a time multiplier which we'll see that we're going to use in our producer, the number of pages that we want to produce, the number of visitors that we're going to simulate, and where we're actually going to produce this file. And keep in mind that the path that I have specified here is where my Vagrant box has a shared directory so that way I can actually reference the resulting dataset from our VM image. So let's go ahead and create a class that's responsible for reading this file and knows how to use this using the Typesafe Config library. So we'll create a Scala class, and we'll call this Settings. We'll actually want this to be an object and not a class. An object in Scala is very similar to a class except that it gets special treatment. And that special treatment lies in the fact that Scala actually automatically creates a single instance of this class and guarantees that only a single instance of this class ever exists. So essentially Scala implements the singleton pattern for free for you. So if you ever needed a singleton, this is how you would create it, and it's a very nice way to group common functionality. And this also replaces functionality of having static classes if you're coming from a Java world. I'm also going to copy/paste some code here and then walk through it. So the first thing you need to do is load an instance of the configuration library, and that's provided through a factory method, ConfigFactory. load. And if you don't want to load the entire configuration section, then the load function has a variety of overrides that allow you load subsections of your configuration. And the way I like to code this is to create an object within my Settings object for every top-level section that I have. So this WebLogGen object here will use config and get the configuration section that it's responsible for, which is clickstream. And if we go to application. conf, you'll see that clickstream is a top-level config that we just created. And all we're doing here is loading them depending on their type, so getInt for records for example and getString for the file path and storing these in local variables within the WebLogGen object. So when our program runs, it'll create a single instance of Settings, also a single instance of WebLogGen, and then all the variables within it, very similar to static classes. Now if you see here all these variables are actually marked as lazy vals on purpose. And the reason I'm using lazy vals here is that I don't want Scala to evaluate the value of these properties immediately. I only want it to evaluate the value of the parameter when it gets used. This comes in handy if you have various overrides for your application properties, and you want to ensure that they all take effect before your val is assigned a value. In general, a lazy in Scala allows for that kind of behavior. So now that we have something that understands how to read our configuration settings, let's go ahead and create a new class, and we'll call this LogProducer, also an object. And we'll have our LogProducer here extend from App. This is a quick Scala shortcut to have the class become runnable without having to define a main function. So essentially the body of the class now becomes your main executable. And in this scenario, I really don't care to have a main function with arguments because we're getting all of our properties from our application. conf file using the Typesafe Config library. Let me go ahead and expand this to get a little bit more real estate. And then for the interest of time, I'm just going to copy/paste some code and then walk you through it. So the first thing you'll notice is I created a wlc variable to get a reference to the WebLogGen config object. And then I used scala. io to load the resource files, both products. csv and referrers. csv, so scala. io. Source and then fromInputStream, and then translate that to lines and to an array. So Products here now is an Array of Strings, and the same things for Referrers that has the contents of our files. And we also need a random list of visitors and pages, so we take the number of visitor count from the configuration, create a range out of that, and then map on each item, and create a string called Visitor-, and then underscore replaces the item that we're iterating through. So this underscore here represents the integer value coming in from the range and the map function. And similarly, we do the same thing for pages, so each one of these ends up being an index sequence of string. So keep in mind here that all we're really doing is creating a glorified random number generator that actually represents a clickstream of logs. So we need an instance of a random object that we'll use throughout the program, and then we'll generate a random number that we'll use to increment time, and we'll base that random number based on the number of records that you're asking for. So to some degree, the time increments and the stalls that we'll introduce are proportional to the number of records that you're producing. The rest of this code is really very simple. We're describing current timestamps and then iterating through a range from 1 to the number of records that you have requested, and then we adjust the timestamp based on the multiplier so this adds to the effect that there's something happening sometime in the future. And then we'll also select an action based on a random number whether it's a purchase, add\_to\_cart, or a page\_view. The same thing will select a referrer, and we will determine the previous page if the referrer's actually internal, so we'll match on the referrer. If it's an internal referrer, then we'll actually generate a previous page. And if it's not, then we'll generate the empty string. There's a similar thing going on here for visitor, page, and product. And then we simply generate the line in the format as we discussed earlier. And then based on the time increment that we introduced earlier, which is random based on the number of records that we have in the file, we'll simply introduce a sleep, so we'll print out the number of messages that we sent so far. And I have a multiplier of 60 here because I just don't want this running too fast. So at this point, we should be able to run our LogProducer. So let's expand this again. So if I right-click on LogProducer, I should be able to see the Run LogProducer because our class or our object extends from App, and that allows me to run it. So you'll see IntelliJ output, and you'll see our print message has sent 106 messages, sleeping for about 4 seconds, and then it sends the last bit of messages. So now if I bring up the directory where I specified to save the data, you'll see that I now have a data. tsv. And if I open it up, you'll see that it generated the records that we've requested. And, again, the format just has an incrementing timestamp, a random list of referrers, actions, visitors, and products. And we should have 200 records here based on the number of records that we've specified in our configuration. If you have any problems with this, then it's likely IntelliJ is being fussy about the Scala integration. I find sometimes closing and reopening IntelliJ gets it unstuck, and it tends to give visual indicators of what might be going on. In most cases, you might see the Scala or resources directories not marked as source roots or resource roots respectively. And I've showed you how you can fix that. In the next segment, we'll create our first Spark program that consumes the file we just created, and we'll follow that by applying aggregations to the dataset.

First Spark Job: Demo

Now that we can generate data, let's go ahead and create another Spark package, and we'll call it batch. And then in that module, we'll create a new Scala class called BatchJob and mark it as a Scala object. Before we can start working on some code, we need to add the necessary dependencies to our project. We'll modify the project's master POM file so these dependencies apply to all submodules. Also, I'm just going to copy and paste everything we need here to save some time. I'll scroll through it so you get a glimpse at how it looks, and obviously feel free to pause the video to copy what you need if you're following along. For those of you that have a subscription that allows access to the code, you should just be able to download this. Let's close this and start our program with a def main with args. And in Scala, this returns a unit instead of a Java void. To start any Spark application, we need a Spark configuration of the form of an instance of SparkConf. Now there are a wealth of options you can set to configure and control your Spark program. And we'll look at a few of them as we progress. But at the very least, every Spark program needs a name and needs to know where and how to execute. If you recall, we discussed Spark needs a cluster manager to run its executors. Spark also comes with its own cluster manager, and you have an option of using YARN or Mesos. But because it's built with that in mind, it can also execute locally very easily, and that's what we're going to be using for the most part of this course. Now to set which cluster manager your Spark program is running against, it's a simple matter of setting the master property in your SparkConf object and related properties for that cluster manager. If you look around, you'll see most examples on Spark set this as part of the code, and I really don't recommend doing that because then you're hard-coding the value. And even if you're reading the value from some other configuration, Spark already provides means to configure this upon launch. Having said that, if you plan to run your program through an IDE, then it does make it easier to do so. So taking our own advice, we'll set the master only if we detect that the code is running from within our IntelliJ IDE. Also, as I'm running on Windows, you'll have to point Spark to Windows Utils. You can download winutils from the source shown on the screen, and you can check out the stack overflow topic for some more details on what's going on. Now that we have a Spark configuration object, we can create the Spark context, which is really the center of the universe for Spark programs and where you can initiate the creation of RDDs, DataFrames, Datasets, and other Spark contexts like the Hive, SQL, and Streaming contexts. So let's create an RDD from the sample Dataset we created earlier from the clickstream generator. The Spark context has methods to create RDDs from various sources. So in this case, we'll create an RDD from the text file. If we check the type of the val input, we'll see it's of type RDD of String. I'll take a quick second here to address the laziness aspect of Spark. So far all we've defined is a way that Spark will create an RDD, and we could perhaps operate on it with a few transformations. As long as we're still composing transformations, we're still telling the Spark context and essentially the driver what we want to do. And these are compiled into steps on the driver, also possibly giving the Spark driver an opportunity to optimize these steps in most cases. So let's assume our Spark application is up and running. And perhaps it has a few executors on a cluster. As long as we're applying transformations to an input, the driver's just merrily accepting them and building out the DAG for execution. Once it hits a piece of code that conducts an action against a previously transformed RDD, only then will it take that DAG, finalize any optimizations it can, and move on to actually submitting it as a Spark job that will run in the context of the Spark executors on the cluster. So in our little example here, we still don't have an action called on our RDD. We'll use the simplest of actions, and that's just to iterate over every record with a foreach and pass it a println function. Foreach is a higher-order function that takes another function. Because the inputRDD is of type RDD of String, then foreach expects a function that takes a string and returns unit. You can think of unit as equivalent to void in Java. So let's go ahead and run the job and give this a try inside of the IDE. And that pretty much concludes our first Spark program. We define a Spark configuration object and set the application name, and we also set the master for running inside of the IDE. And notice here that the master is named local with a star (\*) indicating that we want our Spark program to actually use all of the cores available on the host machine. We then simply create a Spark context giving it the configuration object and define a source RDD called input from a text file containing data. And if we run our program up until this point, all the driver is doing is accumulating instructions of what to do. So all it knows is that there's a text file input that it needs to read. And then we tell Spark to take an action on our inputRDD, and that action is of the form of a foreach iterator against every record where inside of each executor, it will perform the println function.

Aggregations with RDD API: Demo

Now that we've created the basics for our batch operations, let's go ahead and actually do something useful with them and start doing some aggregations on the dataset. And in fact, we actually want to translate this inputRDD to be an inputRDD of a certain type. So let's go ahead and create a package called domain. And then we'll also add a new package object. You can think of a package object as a simple namespace structuring of common code. However, it really is very similar to a normal class that is defined as an object. So technically it is a singleton, but it's a nice way to group common things. And in this case, we're going to use the domain package object to carry a set of common classes that we're going to use throughout our program. So the first class that we're going to create is a case class, and it's called Activity. What we aim at achieving by defining this case class is to be able to operate on an actual object instead of just strings from the input. So let's go back to our BatchJob and see how this works. So I'm going to go ahead and delete this. And then I'll create a new inputRDD which takes the original input and then maps against that. And in here map is a multiline map, and it takes a lambda function. So we'll call each input line, and then we'll take each line, perform a split on tabs, and assign that to a record variable. And then because we're performing aggregations, let's go ahead and define a MS\_IN\_HOUR, so this is essentially just a number of milliseconds in an hour. And the idea is that we'll use this to translate the timestamp coming from the original dataset to an hourly timestamp. And then let's not forget to import our domain. So let's import domain, and. \_ is similar to \* in Java. Going back in here, now I can do something like this, so I can call on the Activity class. And, again, no new keyword because this is a case class. And then we'll go ahead and use record of 0. We'll convert that to a Long. The first field is actually our timestamp, so essentially this will zero out the hours and give us the timestamp by hour. And then we'll just go ahead and apply the rest of the fields. So that gives me an instance of activity. Now if I actually go to check on the type of inputRDD and press Alt+=, it'll show me that it's an RDD of type domain. Activity. So now I actually have a typed RDD of type Activity, which I can use and further manipulate. But let's solve for one more problem here. What if the actual input was not of the length that I expected? So we can check on the length, and I am actually expecting seven fields. And then if it's actually seven, then we'll return an activity. Else, we'll return something else. Now instead of returning a null here and having to check for null exceptions and stuff like that, which is very Java like, let's use something that's very Scala like and something that Spark also uses. So instead of map, let's perform a flatMap. And flatMap expects a type that it will actually unbox. So let's make the result of our if/else statement an option of type Activity. So in the case that we have seven fields, we'll actually provide Some of Activity. In the case we don't have that result, we will return None. And so now essentially what happens is that if the line. split returns seven fields, then we'll create an instance of Activity and return that in a Some. flatMap will unbox it and return the Activity. In case it doesn't, we return None. flatMap unboxes None and essentially drops that record on the floor. So, again, the data type of inputRDD is still RDD of Activity. However, we've been able to apply a map and filter type operation. So now I have an Activity record. However, I want to be able to perform some aggregations by product. So we'll call this step keyedByProduct, and then we'll use our inputRDD. And then Spark provides a convenience method called keyBy, for which it takes a lambda function of your data type and returns K. And K is really up to you what you want it to be. So we'll take an Activity a and return a tuple of a. product and a. timestamp\_hour. So that essentially is our key. So if we look at the signature of keyedByProduct now, it has a signature of an RDD of a tuple of two items, the first item is also a tuple of String and Long, and the second item is actually the entire Activity. Now I know ahead of time that I'm going to be using this keyedByProduct RDD in multiple forks in my code, and each one of those will result in actual action. And so that means it's a very good place to tell Spark to cache the RDD. Caching is a very good idea whenever you have a fork in your code and where you know that re-computing this RDD would actually be very expensive. In our case, this is actually not too expensive, but we'll just do this to illustrate the concept. So the first aggregation and calculation I want to perform is actually a visitorsByProduct. So essentially how many unique visitors for every product has our website seen? So we'll use the keyedByProduct, and then we will mapValues. All I really need here is a. visitor. And then Spark also provides a convenience method called distinct, so it'll actually take a distinct of all of the values per key in your dataset. And then all we really wanted was the count of the distinct. So we'll do a countByKey. Again, this is a convenience method. You could've also implemented this using a normal reduceByKey operation. And, actually, let me show you the data type of the visitorsByProduct per operation. So if we just mapValues, it's still an RDD of the key and string because we translated the actual activity to just the visitor, which is a string. And then if we apply distinct, it'll still be of the same data type because all it's doing is taking a distinct of those values. And then if we countByKey, countByKey is actually an action now. And so that results in a map, a Scala map, so essentially all of the results are actually going back to the driver. So this is a Scala map of a tuple of two, the first being the key, which is a String and Long, and the second being a Long, which is the count of the distinct values. So what we're doing here is counting the unique number of visits per product per hour. So that's a good indication of site activity. However, it doesn't really tell me how much a visitor actually did. So let's go ahead and try and calculate that. So let's call this new variable activityByProduct. And, again, notice how we actually start by the keyedByProduct RDD, and so that's why it was a good idea to actually cache it. And then I know I don't want to do anything with the key, so let's mapValues here again. And it's a multiline lambda, so we'll replace the parentheses with the curly. And then for every activity, let's actually figure out what type of activity this was. So we'll perform a switch or a match statement in Scala against the actual action. Action is simply a string, and match performs like a switch in other programming languages. So we'll say case "purchase" then return a tuple of 1 and 0, 0. So essentially I'm creating a tuple of three values, and I'm applying the first for cases when it's a purchase. And I'll do a similar thing for when the value is actually add\_to\_cart, and also when it's a page\_view. And we'll change the bits accordingly. So now if I take a look at the data type of the activityByProduct, the key remains the same, and I have my tuple of three int, int, and int. So let's go ahead and sum up the results, so we'll do a reduceByKey. And so you see the reduceByKey is operating on the values, so it expects a function that takes two values, each as a tuple of three, int, int, and int, and returns the same data type. So we'll create a lambda function that takes a and b, which represent the two tuples at the beginning, and then we'll just sum up the first field from a with the first field from b and return that as the first field, and then the second field from a with this second field from b, return that as a second field in my result. And then also sum up the third field from a with the third field from b essentially performing a reduction. So now I have the visitorsByProduct, which represents the number of unique visitors by product by hour, and I also have an indication of the level of activityByProduct, which represents the number of purchases, add\_to\_carts, and page\_views by product by hour. So to conclude this program, let's go ahead and just do a foreach and println on these two items. And then we'll save and give it a go. So let's take a quick look at the results. So you'll notice that the first set of records that show up are every product and the timestamp and then the number of unique visitors. And then if we scroll down a little bit, we'll be able to see the second result set, which you'll see here it starts printing out where it's essentially also the product and the hour that remember that was our key, and then the activity whether it's a purchase, add\_to\_cart, or a page\_view. And you'll see some of these actually have 3 page\_views, 0 add\_to\_carts, 0 purchases, and so on and so forth. In the next segment, we'll actually take a look at replicating this logic by using the Spark DataFrames API and actually writing SQL. In fact, the end product of our batch operations will actually be a combination of some SQL and RDD API work.

Aggregations with DataFrame API: Demo

In the last segment, we've seen how we can use the RDD APIs to perform aggregations in Spark. In this segment, we're going to use Spark's DataFrame API to perform similar aggregations. The first thing we're going to need is to import the SQLContext. And then we'll actually need to create a SQLContext from the SparkContext that we had earlier. And then we'll also make this SQLContext implicit. Now in order to perform some operations within the DataFrames API, we're going to have to import some implicits. So we'll import org. apache. spark. sql. functions to demonstrate some function calls. And we'll also import sqlContext. implicits. If you remember what we discussed with implicit conversions, this essentially brings in a class that provides a lot of those implicit conversions for us. Now that we actually have the implicit conversions in scope, we can actually take an RDD of a type where the type is a case class and actually convert it to a DataFrame. Now this function actually comes with two overloads, one where you can specify the column names if you need to, and one where you leave it up to the function to determine the column names. Because we're using a case class, it can actually deduct the column names for us from the case class's fields. So now the import RDD is actually a DataFrame. So let's change its name to reflect that. And then let's actually use some DataFrame functions just to demonstrate how they work. So we'll start off with our inputDF, and then we'll select, and you'll see that select takes a list of columns, and then how about we select everything from the source but do something special with the timestamp. So let's call a function add\_months. This is a DataFrame function which you'll see that this import now becomes active because we're actually using something from that import statement. And if we look at the definition for add\_months, it also takes a column. So we'll provide it with the column inputDF of timestamp\_hour. And then we'll add one month. Now obviously we could have left it at that. But let's go ahead and make sure that the result of the add\_months operation comes up with nice column names. We want the result of this to actually be called timestamp\_hour again. So let's quickly review what we've done so far. We took an inputRDD, performed a flatMap to filter out values, converted the inputRDD to an RDD of a case class type, which is Activity, and then converted that to a DataFrame. And then we used DataFrame operations and column extractors to return another DataFrame with the same columns as we have before but with the timestamp\_hour a month ahead. So now that we actually have a DataFrame and Spark actually understands the types of the fields, we can actually take this and register a TempTable, and let's call it Activity. And by doing so, we can now actually run SQL statements against this temporary table called Activity. And, again, because I know I'm going to be performing multiple operations on this DataFrame, I'm going to cache it. So now if you remember the visitorsByProduct aggregation that we did, let's go ahead and do that using SQL. So val visitorsByProduct, save me some typing. We'll use the sqlContext, and we'll use the SQL function, and then we'll simply write a SQL statement, SELECT product, timestamp\_hour, and COUNT(DISTINCT). This is essentially the aggregation that we actually did in the previous segment. We'll do DISTINCT visitors. And then let's actually give this a proper name, so we'll call this unique\_visitors. And then I also realize that my text is going to wrap off the screen, so I'm going to use a triple quote for the string so I can actually do multilines here. Triple quotes is actually a Scala feature, not a Spark feature. So we'll continue and say FROM activity. This is the name of the table that we defined as a temporary table up above. And we'll simply GROUP BY product and timestamp\_hour. The result of a SQL operation is actually another DataFrame. So at this point, I no longer need all of this. Now because visitorsByProduct is actually a DataFrame, I can actually print its schema. So I can use the printSchema command to see the results of the schema for my SQL statement. So let's do the same thing for activityByProduct. And for brevity, I'm just going to paste some prebuilt code here. So let's take a look at this. Again, it says normal SQL statement goes off of the activity table. And then we select product, timestamp\_hour, and then a normal sum. This represents our reduceByKey here. And then case when action = "purchase" then 1 else 0 as purchase count, and so on and so forth for the add\_to\_cart and the page\_view\_count. And we also group by product and timestamp\_hour. So, again, I can also get rid of all of this. You see how working with DataFrames and SQL statement is actually a lot more clear in the intent as long as the operation actually fits this type of scenario. So what else can we do with DataFrames? Well, there's actually a lot we can do, but let's showcase an example of creating UDF. If you know how complicated it is to create a UDF in Hive or Pig, you'll really appreciate this. So how about we create a user defined function that's responsible for defining underexposed products. So the definition of the user defined function uses the sqlContext. All we need to do is say udf. register. We'll call the function UnderExposed, and we'll define what it takes. So the function takes two Longs. We'll actually give them a name called pageViewCount and purchaseCount. So this is actually the definition of our UnderExposed function. And then all it does is define the underexposure as a percentage of the pageViewCount divided by the purchaseCount. I'm actually going to collapse this to give us more real estate. So now I can actually register another TempTable using our activityByProduct, and we'll register a TempTable. We'll call this activityByProduct. And then I'll just paste a little bit of code here. So we'll call this new DataFrame underExposedProducts. We'll use the activityByProduct table. And we'll call our function that we defined earlier up top here called UnderExposed. So the definition of UnderExposed takes two parameters. We'll give it the page\_view\_count and the purchase\_count. And then we'll call this as---we'll rename this column and call it negative\_exposure. And then the rest is very simple. We'll just order by the negative\_exposure descending and take the top five. I actually just noticed a problem up top where the add\_months function only accepts a date type, and the timestamp\_hour that we have is actually a Long in the form of a UNIX epoch. So we'll use another function that's also provided from the DataFrames API called from UNIX time, which will convert the UNIX timestamp to an actual time. Let's go ahead and give this a whirl and see what happens. So we get an exception. So I was actually expecting this exception because I did something on purpose. The exception actually says that it cannot resolve the visitors column from the input DataFrame. So although DataFrames are great and allow you to use SQL, it doesn't actually protect the developer from making mistakes like this. So all we really need to do here is call this visitor. Now I actually bring this up on purpose because there's a new API called the Dataset API which is very similar to the RDD API we looked at earlier but gives you added benefit and performance of a DataFrame. So Spark actually understands what you're doing within the RDD, within your lambda function. Earlier in an RDD API, Spark had actually no insight into your lambda functions. With the Dataset API, Spark understands what you're doing in your lambda functions but also gives you the added benefit that there are types for the columns. So it looks and feels as a DataFrame to Spark, but it also looks and feels as an RDD API to you. So you kind of get the best of both worlds. And it is Typesafe. So while it actually doesn't provide you any mechanism to fix SQL mistakes if you're using SQL, but if you are doing other types of operations, you'll get the added benefit of performance as if you were using a DataFrame while still looking and feeling like an RDD API. If you run this yourself, feel free to examine the results, but they're not very important because, again, we're basing it off of a random record generator. So the results really don't mean much apart from verifying that your Spark program is actually doing what you expect it to be doing.

Saving to HDFS and Executing on YARN: Demo

Now that we've seen how to perform aggregations with DataFrames API and the RDD API, let's move one step closer in these steps that we want to achieve for our batch operations. So let's go ahead and start with the BatchJob Spark application again. And we're going to be taking a few steps here. The first step is that we're going to be start preparing our job for actually submitting this on the YARN cluster using the spark-submit. And we'll also be doing a little bit of cleanup on the way. And the second part is that we are going to examine Spark's data source's API and actually write data out into HDFS. So the first thing we have to do to prepare the job for submitting using spark-submit against a YARN cluster is to fix our paths. If you look at our current setup, we're assuming that we're actually running local, and our paths are all assuming a local file system. So when we run this using spark-submit on YARN, it also needs to understand how to get to the data. If you recall, this was the directory I used when we first set this up. So this is where I did the Git clone, and this Vagrant directory represents the shared directory that the VM sees. Even though this is the shared directory that the VM sees, our Spark setup actually looks at it a little bit differently. So the first we have to do regardless of our Spark setup is actually prefix this with the file system type. So we're still going to be using a local file. So we'll start with file. And then our YARN cluster is actually set up to understand this shared directory, but it's mounted directly under root. So it will be /vagrant and then, again, UNIX file system here so vagrant/data. csv. So that should get us going for the source file. The next thing I want to do is just remove a little bit of clutter. The UnderExposed function that we used and created in the previous segment, we're not going to actually use that in the remainder of the course. So let's go ahead and remove that and also remove some of the dependencies that rely on it. We don't need the underExposedProducts calculation anymore. And I also want to clean up this printSchema. So the next step I want is to store activityByProduct onto HDFS. So we'll store the result of this select statement onto HDFS, and we'll also still print it out. To do that, we're going to be using Spark's Data Sources API. The Data Sources API in Spark operates on a variety of data sources that are provided through the DataFrame interface. So we'll start with the DataFrame, and each DataFrame provides a read and write generic method. And then we can select the mode of operation. In this case, it's a Save mode because we're actually writing data. And we'll select Append in this case. Any by choosing Append, every time you run this program, it will add data to the location that we choose in a second. And Spark's read and write operations actually have a default of parquet. But I like to be specific just in case that default actually changes down the road. And then we'll provide a fully qualified path for where we're saving the data. So this will be lambda-Pluralsight, port 9000. That's where the VM is running the name node. And then in our VM setup, there's actually a lambda directory that is globally writable that I provided for you. So we'll write under lambda, and we'll call this batch1, which will ultimately create another directory. So we could stop here, but given this is a timed data source, let's go ahead and actually partition our Datasets based on time. So Spark also provides a convenience method, whereas we can partition the data. So we'll partition by the timestamp\_hour column. So if you're familiar with Hive and how it actually does partitioning, whereas it has a directory with the name for each value of that partition, that's exactly what Spark is going to do here and create subdirectories for each value of timestamp\_hour. And, finally, we'll clean this up as well because we don't need it. Before we run this, let's take a look at what we have here. So we're defining the activityByProduct DataFrame as a result of this SQL statement. In Spark terms, this SQL statement is a transformation. So there's no action yet on activityByProduct. What follows is a write, which is actually an action on the activityByProduct DataFrame, which will result in a driver materializing the DAG and sending it to executors to perform its execution. But we also use activityByProduct in a foreach here to print the results. So that means the driver's actually going to perform this SQL statement twice--once for actually writing the results and once for the print statement. And it'll do that because we don't have it cached. So this DataFrame isn't cached. The activityByProduct DataFrame or the result of the SQL statement is not cached. It'll actually go all the way back to where there is a cache or it hits the actual original data source. And in our case, it'll actually go back to this DataFrame here because this DataFrame was cached. So to save us a few steps and a few trips, we can go ahead and cache this. And now our code is ready for primetime, so let's save. And then we will package it. After IntelliJ is done packaging the code, you should end up with a fat jar, which contains all of your code and all of the dependencies. And that's all controlled by how we set up our POM file here. Now that the packaging completion is finished, I need to grab the correct jar. And you'll find that under the target directory of the project we're in, so we're under spark-lambda. Under the target directory, you'll see a spark-lambda, and we want the shaded jar because that includes all of our dependencies and all of our configures. So I'll go ahead and right-click and copy this file. And then I'm going to place it in the Vagrant shared directory. This is the same directory that we have our data file in and the same directory that is shared with the virtual machine. So we'll be able to pick up our jar from there. Now to submit our application, make sure that the virtual machine is running and go ahead and bring up Cygwin. Once you have Cygwin up, go ahead and navigate to the Vagrant box shared folder. And then if I ls in here, I should be able to see my application jar making sure I dropped it in the right location. So let's go ahead and vagrant ssh into our box. And then you should also be able to see the jar from within the VM. And then also on the VM, there's a directory called Pluralsight and then a Spark directory right under that. This is a local build of Spark with Scala 2. 11 and Java 1. 8. Now to submit a Spark application, we'll use spark-submit under the bin directory. And spark-submit takes a few parameters. The first we're going to provide is for the master. In this case, we're going to provide YARN as the master since we're going to be running this on our YARN cluster. And then we'll also specify deploy-mode as cluster and then the class name. So our class was under the batch module, and it was called BatchJob. And then, finally, the path to our application's jar. So you'll see that the application starts shipping the jar to the YARN cluster, and it starts executing. We'll actually monitor the application using our browser. So go ahead and point your browser to localhost port 8088, so that's localhost port 8088. So if you refresh this, you should be able to see your application running. In my case, this was my second attempt. And if you click on the application master, that actually takes you to the Spark UI where you can monitor the jobs, stages, and executors. So I'm just going to hit F5 until I start seeing jobs executing. So this is the parquet at BatchJob Scala line 64. We'll see it's actually executing 0 of 2, and you can even expand this and visualize the DAG that is actually going to execute. And you'll even notice some green dots here that indicate that that's a point that we've defined for Spark to cache our dataset. And you can see that's evident by the stage and the operation running here. So it seems like our job actually finished. So go ahead and point your browser also to lambda-Pluralsight or localhost. Both should resolve to localhost and port 50070. That brings up the namemode UI. Under that, there's a Utilities, Browse the file system. And then if we simply specify /lambda, we should be able to see the batch1 directory that Spark created for us. And then under that, you'll see that the data's actually partitioned by the timestamp. And under that, you'll see all of the parquet files. Now the number of these parquet files actually map to the number of partitions Spark was actually using to execute your job at the point when it was writing to HDFS. And if you know anything about Hadoop and HDFS, many small files, especially in the 1. 71 KB range, is not a very good idea. So we could have solved this problem if we wanted to by simply using a coalesce function on the Spark DataFrame or RDD before the write operation to HDFS. So with coalesce, you can define the number of partitions you want, but it does come with some cost because it does require a shuffle. This is very similar to forcing your reduce phase in MapReduce and selecting the number of reducers. Finally, just for completeness, we could have looked at these files from the command line. So we could have simply used hdfs dfs - ls and then looked at lambda. And you'll see there's a batch1 directory. And if we look under that, we'll see the same data listed. So far we've proven that we can take and put data, do some aggregations on it in Spark, and then write it out to HDFS. So how about we read that data back in Spark using Zeppelin. In the next segment, we'll use Zeppelin to look at Spark's Data Sources API and read the parquet data back into Spark and run a few queries.

Querying Data with Spark DataSources API: Demo

In this demo, we're going to use Zeppelin to create the data that we saved into parquet files from the previous segment. Go ahead and point your browser to localhost port 8988. And we're going to create a new note. We'll call it batch1. So we're going to be using Spark's Data Sources API to read the data out of HDFS. So we'll start off by creating a val. Let's call it parquet DataFrame (PDF). And then Zeppelin already provides an instance of the sqlContext, and the Data Sources API provides a read method, and then also a format method to define the format of the data that we're going to be reading. In this case, it's parquet. And then the common load method, which in turn takes the path. Now there's also a shortcut specifically for parquet where you don't have to provide the format and load, and you can simply specify parquet, very similar to what we did when we actually wrote the data out. Now DataFrames also provide a convenience show method. So we can use that to print out the schema and a few records. So Shift+Enter to execute. You'll see that it actually understood the fields that we have defined in our parquet files, and it can actually read out the data, and it prints the schema. Now let's say I wanted only the records that had a page\_view\_count of greater than 2. So I can take my parquet DataFrame, register a TempTable, let's call this source. And then I can use the sqlContext to query the TempTable, so SELECT \* FROM source where page\_view\_count greater than 2. So while this works and returns another DataFrame, let's call this filtered, it's really unnecessary now because Spark actually provides additional convenience methods where you can perform queries like this against source files without even having to register a TempTable to do that. So back to the top. And then instead of the read, we're going to use a SQL statement. So sqlContext. sql. And then instead of immediately defining the path or starting with that, we'll actually start out with the query. So we'll SELECT \* FROM and then the keyword parquet gets special treatment with the dot and then back tick, then providing the path, also closing the back tick. And we continue with our WHERE clause page\_view\_count greater than 2. This syntax looks a little bit unfamiliar and unnatural, but if you know a little bit of Scala, then you understand that function names can have a variety of special characters. So this is actually a call to a function that Scala understands, and it feels to you as if it's a DSL. So now keep an eye on the previous records that had page\_view\_counts of 1. If we re-run this, those should disappear, and indeed they do. So let's run a couple more queries. So we'll keep this table registration, but we'll remove this filtered query. So go ahead and execute this to register the TempTable. And now instead of using sqlContext. sql and providing the SQL statement, let's actually have Zeppelin understand what we're doing. So we'll provide it the SQL binding and just by doing that, it understands that I'm about to write Spark SQL. So how about we figure out if there's a relationship between the length of a product name and the total number of page views. So we'll sum up the page\_view\_count, call this page\_views from source, that's the name of the table that we've given, and then we'll group by length of product. Shift+Enter to execute. So it's not very obvious just by looking at this, so let's actually display this in a chart. Interesting. It looks like there's a sweet spot between 21 and possibly 33 characters for your product name. Now I will reiterate that this is random data so the results really don't matter. But it was an exercise to see how we can use the Data Sources API to save and query data using Spark. And while we haven't integrated this into our code just yet, it will help us put everything together and glue the pieces when we finish up working on our lambda architecture.

Summary

Let's summarize what we've covered so far and relate that to the lambda architecture we're building throughout the course. In this module, we created the first version of a LogProducer that generates web logs and uses the Typesafe Config library for the producer's configuration. In subsequent modules, we will continue to modify the LogProducer to send a stream of messages to Kafka from which we will have a Spark Streaming application consume from. In this module, we also created the first draft of our batch layer implementation with Spark. And now we have some experience creating aggregations with Spark's RDD and DataFrame APIs. We also used Spark to save data in parquet format on HDFS and then queried that data using Spark SQL and Spark Data Sources API. In the following module, we'll introduce the concept behind Spark Streaming and then also implement the first draft of our streaming application. We'll learn how to create stateful aggregations in Spark Streaming, and we'll refactor our application to start sharing code between the batch and streaming layers realizing the benefits of using one framework, Spark, as the fundamental backbone of our lambda architecture.

Speed Layer with Spark Streaming

Intro

Hi! This is Ahmad Alkilani, and welcome to this module. In this module, our focus will shift to building the speed layer in the lambda architecture using Spark Streaming. Let's have a look at what we're going to cover. Most of this module is dedicated to Spark Streaming, so we'll cover Spark Streaming fundamentals and give you a solid understanding of how streaming works. So when you're working with a streaming application, you know how to navigate and relate to what you've learned earlier with RDDs. We'll also show you how you can combine Spark SQL and DataFrames with your streaming application. And we'll briefly discuss and introduce the streaming receiver model and how Spark collects data. This will become critical when we start working with Kafka, especially when we discuss different approaches to receiving data. However, as we're not covering Kafka just yet, we'll also modify our log producer to simulate streaming data into files, and then we'll work with files directly. As we continue to build our streaming application and aggregations in a streaming fashion, we'll also work on the overall structure of our program. We'll separate the business logic from spin-up and teardown procedures for cleaner code. Finally, we'll look at how we can run and test our application using Zeppelin and what it takes to get a streaming application to work with Apache Zeppelin. In the next clip, we'll start off with how Spark Streaming works and introduce DStreams, so let's get started.

Spark Streaming Fundamentals

Spark Streaming is an extension of the core Spark API. It enables building scalable, high-throughput, and fault-tolerant streaming applications, all without working with a vastly different toolset and staying within Spark. If you think of Spark as a platform for working with data in batch, then Spark Streaming extends that concept to work with data in much smaller units but in a very similar fashion. We call this micro-batching. Spark Streaming can work with a pretty impressive list of prebuilt streaming data sources, Kafka possibly the most important one, and we'll discuss Kafka in more detail in the next module. You'll also notice AKKA has a different shade, and that's because that's the most raw form of a stream provider. Spark provides the necessary interfaces and hooks for you to build your own streaming provider using AKKA as the messaging platform. Spark is actually implementing using AKKA, so it only comes natural to provide something like this. In any case, Spark comes with a long list of prebuilt sources in addition to community-provided sources that you can adopt and provides means to act on the data received. Examples would be to save the results out to HDFS or external data sources like oracle, SQL Server, Elasticsearch, Cassandra, and the list goes on. At the core of Spark Streaming is a class called DStream. A DStream is merely a collection of RDDs with time information. It also comes with additional functions, for example, to maintain state as your streaming application progresses and also to provide a means for windowing calculations. Let's take a closer look at how Spark Streaming works. We obviously need some input data stream that feeds into a Spark Streaming application. There's obviously something that needs to happen for Spark Streaming to get a hold of this input Dataset. But we're not going to focus on that for now. So let's table it for later. For now just note that there's a process denoted in the slide here by the vertical dotted line, and this process is responsible for providing the data. And there's also something called a block manager in Spark that keeps track of this data as it lands in the Spark's universe. In every streaming application, the application has to define a batch interval. This is essentially the micro-batch interval we were discussing earlier. The shorter the interval, the lower the latency between input data being available and processing against it begins. However, that doesn't necessarily define how long it takes for the processing to complete. Ideally, processing completes within each batch interval. And the interval simply acts as the trigger to launch the same processing steps against the next batch of data. However, if processing takes longer than the batch interval, then processing queues up. And if that remains the case, then your application is deemed unstable and will eventually fall over. The point here is that there are more implications to selecting a batch interval than just a control of latency. Your processing needs to finish within that time. And the best way to know that is to test and resolve any bottlenecks in your code by looking at the Spark Streaming UI. As your data streams in, it's placed into blocks or partitions. There is a Spark configuration parameter called spark. streaming. blockInterval that controls the interval at which data received by Spark Streaming receivers is trunked into blocks or partitions. So partitions are time-based or time-bound and not size-bound. The collection of partitions in a similar time-bound fashion based on the batch interval collectively creates an RDD. Once an RDD is created from a batch, it is queued up and scheduled for execution by the driver, which then very similar to any normal Spark application schedules the execution of your code, transformations, and actions against the Spark executors where each task is responsible for handling the work of a partition. And normal operations continue here as expected. As data's being processed, more data is received and accumulates into partitions, which then become another RDD and so on and so forth. Now to the programmer, the entire collection of RDDs is represented as an abstract called a DStream, short for discretized stream. A DStream is a class similar to an RDD but with a few additional functions to support streaming-specific operations like windowing and maintaining state. Also notably, DStreams miss a few functions that RDDs have, but we'll see shortly that there are ways to work around that that are built into the DStream, specifically the ability to drop down from a DStream to the underlying RDDs to do our work. So to summarize, your application defines the batch interval, and a Spark configuration defines the time to wait to establish a partition. The driver's job then apart from keeping track of this data is to take your application's code and execute the Spark transformations and actions against your data. And this continues to happen in a micro-batch-type fashion achieving the streaming that Spark supports. Now this approach doesn't come without its shortcomings. It's obviously not acting on a single record at a time as when it receives it. That's not to say you can't operate on each record. It just means there is latency involved that can be up to the batch interval in a healthy system or even higher in systems that are struggling to process data as fast as it's received, for example, under certain bursts. The point being is it's not a complex event processing engine, and latency in processing data is to be expected. Having said that, there are definite wins to this approach. Because it relies on RDDs under the hood, at least once, at most once, and guaranteed delivery of data are not only possible but are much easier to achieve when comparing Spark to other systems. Spark, for example, already has fault-tolerance because it stores the lineage graph of operations. Another advantage is throughput. Think of Spark as a bus transporting people, each person representing an event. Other streaming systems that operate on data in a continuous operator model or single event at a time fashion can be thought of as a car carrying a single passenger. A single car travels more nimble, but put 5, 000 cars on the road at the same time, if you're not processing data fast enough, all of the sudden your system's congested. Compare that to a bus, you still have a lot of free space on the freeway. Finally, one advantage that may not be as obvious as the others is the fact that Spark Streaming uses the same core. So under the covers everything is RDDs. That means familiar programming constructs, code reuse, and lower total cost of ownership. And any advancements in RDD behavior and performance also reflect on Spark Streaming. That's what makes Spark and Spark Streaming such a compelling case to implement a lambda architecture with.

DStream vs. RDD

We noted that a DStream has similar functions to an RDD, so how closely related are they? Let's start with a DStream of type T. A DStream has a private generated RDD's variable, which has a type of HashMap of Time and RDD of the same type T. So each DStream keeps a HashMap of all the RDDs it composes from. But it also keeps track of the time at which each RDD was made available. Streaming applications can run for a very long time, so keeping track of every RDD that passes is actually unnecessary. DStreams only need to remember a portion of that lineage that still pertains to the operations at hand. So there's another private variable called rememberDuration that tells a DStream how long it needs to remember RDDs. The generated RDDs HashMap and its memory footprint are affected by a list of things, for example, window calculations will cause the duration to expand. You can also manually request for a longer duration if you think Spark isn't doing a good job at keeping RDDs around. But it tends to do quite well. Also, a Spark feature called checkpointing is typically used to checkpoint the lineage graph and RDDs alike, so that typically allows DStreams to discard older data from memory reducing memory footprint. So DStreams have collections of RDDs. But they also share a very similar API. There are transformations, your typical map, flatMap, filter, reduce, and union. And then we have another class of transformations specific to window operations like window, reduceByWindow, and reduceByKeyAndWindow. And, finally, DStreams have a set of transformations that are responsible for maintaining state, specifically updateStateByKey and mapWithState, which we'll be looking at in a lot more detail in our demos. DStreams also have output operations similar to actions in RDDs. Examples are print, saveAsTextFiles, and saveAsHadoopFiles. Finally, there are two very special functions that allow you to drop down from a DStream and operate on the RDD within the DStream for that batch. The functions are called transform, which comes as a transformation, and the second called foreachRDD, which comes as an output operation or action. Transform and foreachRDD can be used to apply an RDD operation that is not exposed in the DStream API. A simple example is being able to join a data stream with a static Dataset represented as an RDD. Let's further examine these two functions.

Using transform and foreachRDD

The DStream transform function is a transformation on a DStream that allows you to drop down to the RDD and operate against the RDD API. If we look at the default function signature for transform, it's a higher order function like map, which means it takes a function. The function is called transformFunc. And this function is supposed to accept a single parameter of type RDD of T and produce a result of type RDD of U. So that means you get an instance of the underlying RDD, and you can transform that RDD to an RDD of any other type. An example would be an input DStream of String, transform on that, and you'd get an RDD of String to work with. Now you may choose to convert the string to an instance of an object, let's say Person, and return an RDD of Person. So the resulting DStream after transform is a DStream of Person. Here's another practical example. We have a DStream called clicksDStream, and we want to join the RDD from each batch to another RDD called filterDomainRDD, which contains a list of domains we should exclude from our results, so essentially acting as a filter. So we use transform on clicksDStream and provide a function, in this case a lambda function, that takes an RDD and returns another RDD that results from us joining against the filterDomainRDD and performing a full transformation. The resulting DStream in this case called cleanedDStream ends up with the same type and unwanted records filtered out. The foreachRDD operation is very similar to transform except the function provided returns Unit, which is roughly equivalent to void in Java. So the function for each func acts as a sync for the data records. A great example of this is saving data to Cassandra. DataStax, the main company behind Cassandra, has built a library that works with Spark to easily save and retrieve data from Cassandra. We'll be looking at this in detail in module 5, but this is a great introduction. The Spark Cassandra library provides implicit conversions to extend the functionality of an RDD. So essentially the Pimp My Class pattern. Upon importing the library classes, a normal RDD will appear as if it always had a method called saveToCassandra. But the save operation was only implemented for RDDs. Using it from a streaming application is simply a matter of dropping down to the RDD using foreachRDD and calling saveToCassandra on the RDD itself. By the way, if you skipped the Scala introduction in module 1, we explained Scala implicit conversions there, so this might be a good time for you to catch up on that.

SparkSQL in Streaming Applications

We've been talking about DStreams and RDDs, but whatever happened to DataFrames in Spark SQL? Well, you can still use them by applying either transform or foreachRDD. So you would drop down to the RDD level and use the RDD API to get a DataFrames or use the SQLContext. Here's a good example from Spark's own documentation. It assumes a DStream of String called words. foreachRDD is then used to drop to the RDD level. And a SQLContext is created, and its implicits are imported similar to what we've done in the previous module. The RDD is then converted to a DataFrame of a single column called word, and a TempTable is registered as well. Then we can simply write SQL against the table and, yes, you guessed it, this is yet another word count example but with a bit of Spark Streaming this time. In the next clip, we'll discuss Spark receiver model before we start working on the code for this module.

Streaming Receiver Model

Before we get into the nuts and bolts of our code, there's one thing that we did not cover, and that's Spark's receiver model. Or in other words, how does Spark consume data and present it as micro-batches in a streaming fashion? This is the discussion that we deliberately skipped when we first started discussing Spark Streaming architecture. Let's take a simple example where I have an input data stream, and let's say this was a stream of data coming in through the network on a socket. Spark has a prebuilt function to handle reading strings off of a socket called socketTextStream. In our example here, we define a lines1 variable that reads strings off of port 9999 on localhost. Lines1 is an input DStream, and most DStreams in Spark (however, not all of them) use something called a receiver, which is an object that consumes a task and is responsible for receiving data from a source and storing it in Spark's memory for processing. There are a few interesting things going on here worth noting. The first is that a receiver occupies a task all for itself and, in fact, always maintains occupancy so the task assigned for the receiver is ultimately always occupied by the receiver essentially eating up the CPU core. That's not necessarily a bad thing if your data is high volume. But in cases where your data rate fluctuates significantly, you may want to free up that core for other tasks. In a receiver-based approach, you simply cannot do that. That task is designated for the receiver whether it's doing something or not. So for a receiver-based streaming application, you need to at least allocate one core for the receiver, assuming you have only one receiver, and one core for your driver, and a third for task processing. Those are the minimum requirements. The second note is that the maximum amount of input parallelism you can achieve is controlled by the number of receivers you have and, obviously, your input stream's capability of handling multiple streams. You can increase your receiver input parallelism by introducing additional streams. So let's introduce a lines2 stream, which reads off of a different port and see how that looks. So now Spark has two tasks dedicated to receiving data, which is great if you need more parallelism on the receiving end. Notice how this also impacts a number of tasks available for processing the data, so you'll have to weigh these two options especially if you're limited on resources. Now that you have two data streams, how would you operate on them in unison? Surely you don't want to have to write code that processes lines1 and then processes lines2 all over again. Spark actually comes in and has a simple solution for that. You can create another DStream that is the union of your input DStreams and then operate against that. This union operation is merely a meta-operation, so you don't incur any additional costs by doing this. It just impacts the programming model and does not end up shuffling data all over the place. The third note here is that of resilience and reliability against failures. Spark's receivers are already resilient against failures. If the task designed as a receiver dies or the entire executor dies for that matter, another task is assigned as the receiver, and life goes on. No big deal. But what about the data that the now-dead receiver had in its memory that it hadn't passed on to Spark as a partition in RDD yet? What happens to that data? Well that ultimately depends on the implementation of the receiver. Spark classifies receivers into two categories--reliable and unreliable. Reliable receivers correctly send acknowledgements to a reliable source when the data has been received and stored in Spark with replication guaranteeing reliability in case of a failure. Unreliable receivers obviously don't carry that trait. From the definition, reliability obviously isn't just a function of the receiver's implementation itself, but also the source stream being deemed reliable as well. A great example of a system that can handle reliability and playback as needed is Kafka as we'll see in the next module. Our little example here with the socketTextStream is not reliable because there's nothing to act back to that can reliably carry the data for us until we restart or recover. Finally, Spark's receiver model doesn't apply to every input source, for example, the file stream source doesn't use receivers this way. Rather, it's a much simpler implementation involving the driver. There's also another approach to streaming that Spark introduced called the direct approach, and it was implemented to handle some of the shortcomings of the receiver-based approach first for input data streams that can help coordinate the effort. We'll discuss the direct approach in more detail when we visit Kafka in the next module. So that covers the receiver model, and I think we've had just about enough slides for this module. Let's move on to writing some code and building a streaming application.

Creating Spark Streaming Application: Demo

We're going to start off our code in this module by doing a little bit of reorganization. I like to organize my code into helper classes where it helps separate program setup from helper classes. And that really gives me the opportunity to reuse this code in different projects. So we'll start off by creating a new utils package. And under the utils package, we'll create a new Scala class called SparkUtils and also mark this as an object. And part of what we're going to do here is create helper functions to help provide the Spark context. So we'll define a function called getSparkContext. And the function will take the appName as a string, and it's responsible for producing the Spark context. And knowing that we've already done that in the BatchJob, we're going to cheat a little and copy some code from here. Let's grab the code from the get\_spark\_configuration all the way to where we defined the SparkContext. And then we'll paste it here and do a little bit of cleanup. So the appName is provided as a parameter to the function. And then I know I'm going to reuse this bit of code that determines whether we're running inside of the Id or not, so I'm going to extract this into a variable. So val isIDE =, and we'll give it this expression. So now I can replace the if statement to use the isIDE variable. And then towards the end of the function, I know I want to return the SparkContext, and I don't want to return the SQLContext in this case, so we'll just return the SparkContext. Let's also go ahead and define another function called getSQLContext, which takes a SparkContext. And then we'll simply create a SQLContext based off of the SparkContext provided. So SQLContext. getOrCreate, and then we'll pass it the SparkContext provided and return the SQLContext. Now there's one more thing that I want to add to the getSparkContext function. I want to add a facility to create a checkpoint in Spark. So we'll define a checkpointDirectory string, and then we'll set the value of the checkpoint depending on whether we're running inside of the IDE or not. So if we're running inside of the IDE, we'll want the checkpointDirectory to point to some local files, so file. In my case, I'll just point it to my F directory in temp. Then if it's not running inside of the IDE, we'll set checkpointDirectory to an HDFS location. So our fully qualified NameNode, Pluralsight, port 9000. And then under spark/checkpoint. Now I will go into more details and explain how checkpointing works especially in the case of Spark Streaming, and we're using it here while we're getting the SparkContext. It actually gets very limited use in normal Spark application. But we define it here nonetheless so you can use it if you want to, and then also when we create the function to get our Spark Streaming context, we'll just piggyback off of the checkpoint directory defined on the SparkContext and use that so you don't have to set it in multiple places. So let's go ahead and continue working on this. So now that we have a SparkContext here, I'm actually going to change this to a getOrCreate. So instead of a new SparkContext, we'll do a SparkContext. getOrCreate and provide the configuration. Now the usage of getOrCreate here isn't just for fun. There's obviously a reason why we're using this. What this does is it helps us have functions where we don't necessarily need to pass around the SparkContext all the time, and they can always call the getSparkContext utility function that we have here. And if the context has already been created earlier, it will simply just return that context from an earlier creation. So this definitely does help a little bit with programming. And now to set the checkpointDirectory, so sc. setCheckpointDirectory. And we pass in the variable. So now that this is cleaned up, let's actually go back to the BatchJob and use the utilities that we have available. So let me just copy the application name from here, and then we can get rid of all of this. So we'll set up the spark context, val sc = getSparkContext, and then we'll provide the application name. Don't forget your import, so import utils. SparkUtils. \_ (that's similar to \*). So now that resolves. And then we'll also create our sqlContext. And there we go, already a little bit cleaner. So now that we have this taken care of, let's go ahead and create another package, and we'll call this streaming. And then we'll also create a new Scala class. We'll call it StreamingJob. And, again, type Object. So only the main function to make this runnable, so def main(args: Array of String). And then this returns Unit. So, again, here let's set up our spark context, so val sc = getSparkContext, and then Lambda with Spark. Let's not forget our import, so import utils. SparkUtils. \_ to import everything in there. So as we know, the SparkContext is actually the center of the universe for a normal Spark application. Similarly we also saw a SQLContext earlier that we were able to use to run SQL functions. And for streaming use case, there's actually something called a streaming context. So we'll create a Spark Streaming context, that's ssc = new StreamingContext, and I'll hit Alt+Enter for the IDE to do the auto-import of the StreamingContext and Ctrl+P to show the function parameters so we see it takes a SparkContext, so we'll provide it that. And then it potentially takes a checkpointDirectory, and we'll get to that shortly. But we're actually going to use a third signature in this example where it takes a batchDuration. So let's define a variable called batchDuration. So this is actually your micro-batch interval. We'll define this as 4 seconds. And then also Alt+Enter to auto-import. We'll provide the batchDuration to the SparkStreamingContext constructor. So now I have an instance of a SparkStreamingContext. So what can we do with it? Well, let's try using it on some of our input files that our LogProducer can generate. And we'll define the input path based on whether we're running inside of the IDE or outside of the IDE. So the variable that we've created will come in handy. Let's go back to SparkUtils and actually move this variable outside of this function. So we'll make it part of the object SparkUtils itself. So now if I go back to our StreamingJob, I can use isIDE, and then we'll perform a simple match against that. In case it's true, I'm going to give it a file path that will resolve if I'm actually running it from my host. So on my host, my Vagrant box share was under this path. So f:/Boxes/spark-kafka-cassandra-applying-lambda-architecture, and this is my Vagrant share. Then I've created a new directory under that called input. We're going to be creating and deleting files frequently, so I'd advise you to actually create this new input directory. If I'm not running this from the IDE, that means I'm probably running this on the VM, so I'll provide a path that will resolve on the VM. So the Vagrant share up here actually resolves directly under root. And then the input directory. That's how the VM actually sees this share. And now we get to create our first input DStream, so val\_textDSstream = ssc. textFileStream, and then we'll provide that with the input path. Now if I look at the type for textDStream, it actually shows that it's a DStream of Strings. Now if I just close and wrap this up here, Spark will actually complain that there's actually nothing for it to do because there are no actions performed on the inputDStream. So let's go ahead and perform an action. So if you remember print was actually one of those actions. So we can simply just print the context, and that will print a number of records. You can go ahead and provide the number of records if you want to here. And now to actually kick off a Spark Streaming job, you need to start the context. So ssc. start. So that will start this SparkStreamingContext so we'll actually start receiving data, creating RDDs off of that with a batch duration of 4 seconds. And for each dataset that is consumed within 4 seconds, it will simply apply our transformation. We're not really doing anything here except the action of printing out some results. Now the only problem with this is that the start function returns immediately, and we need something to keep our RDD or our Spark program alive while it's streaming the data. So Spark provides a convenience function called awaitTermination, and you can either await termination indefinitely or await termination with a timeout. We're just going to wait indefinitely and stop our running job if we need to.

Streaming Log Producer: Running with Zeppelin: Demo

Now before we can actually run this, I'll note that the textFileStream isn't too intelligent in Spark, and it's merely used for very simple scenarios or for tests like this. So all it really can do is monitor a single directory from new files and then consume the data from that file. But it won't keep a watch on the same file to see if new records or new data is added to it similar to what you can do with tail in Linux or with Logstash, for example, as an application. So we'll need to make some slight modifications to our LogProducer. So let's go ahead and go to our LogProducer. And then what we're going to do here is create another for loop to create multiple files. But before we can do that, we need to add additional configuration properties. So we'll go to the application. conf file, and we'll copy this, create a destination path, and that destination path is going to be input. And then let's also define another variable for the number of files that we want to produce, so number\_of\_files. And then let's say 50. So the pattern we're going to follow is that we're going to create a file under the filePath, so we'll create data. tsv under Vagrant. And then once that's created, we'll move it into the Vagrant input directory with a certain name. So to help with these operations, go ahead and open the POM file, and we're going to add the dependency for commons. io. Commons IO is an Apache project that provides file utils helper functions. Let me go back and expand this. So before we go back to our LogProducer and modify the code there, let's go ahead and go to our config settings and add the settings that we just defined in our application. conf file. So we'll create another lazy val and call this destPath, weblogGen, and it's a string so we'll getString, and we named the configuration dest\_path. And we'll also create a lazy val for the numberOfFiles, and this time it's actually an integer that we're getting, so number\_of\_files is what we'll call the configuration. And now we can go ahead to our LogProducer and actually use these. So right under the filePath, let's create a destPath variable, and we'll get the value from the destPath from our web log config (wlc). And then we'll introduce another for loop, so for (fileCount 1 to wlc. numberOfFiles. So we create a range in Scala using this shorthand syntax, and that defines our for loop. So we're going to loop over this entire set all the way until after we close the file. Then after we close each file, we'll define an outputFile, FileUtils, and the FileUtils we want are the Apache Commons IO FileUtils. So we'll get a file. Let's use string interpolation to create this file name. So what we want to do is use the destPath and then append a file name called data\_, and let's actually use the timestamp for the postfix of the file. So we'll timestamp our outputFile names by the timestamp of the last record in each file. And then let's just do some simple print lines to show us some progress so it'll indicate that we are planning on moving the produced data to the outputFile location. And then we'll use FileUtils to actually move the file, so we'll move a file, and our source is a FileUtils. getFile. And we'll provide the source file path, which is defined in our filePath variable. And then moveFile expects a destination instance of the file object, and that was represented by our outputFile up above. So we'll create a variable called sleeping. Let's say we'll sleep for about 5 seconds between the creation of each file and the other. Let's also give a visual indication of that. So I'll say "Sleeping for, and then $sleeping ms. " So now our LogProducer actually creates logs in this format, adjustedTimestamp\t$referrer\t$action\t$prevPage\t$visitor\t$page\t$product. And it does that while sleeping for random increments finally closing out the file. And once that's file created, we create an instance of the file object defined as the destPath and the literal data postfixed with the timestamp just to make these file names unique. And we'll sleep between each iteration. So I'm going to try and run this now, and I'll bring the Vagrant directory here in view so you can see what's going on. So this is our input directory, and you can see that it's empty. And I'm going to go ahead and stat our StreamingJob, so Run StreamingJob. So you can see the job has actually started. And now you can see that it's actually starting to print the output for every batch. So every 4 seconds, which is our batch interval, it's performing an entire set of transformations and actions. And realistically we only had a print statement. And because it's not consuming any data, there's no data being produced to this directory. You don't see anything, but it's still showing that it's printing the results. So now let's go back to the LogProducer, clickstream, LogProducer, and we'll start this. And you'll notice that the LogProducer's running on a separate tab here, and the StreamingJob is still running on the first tab here. So now that the first file---you saw that the first file dropped, and it's actually producing and printing out the results in a streaming fashion. That's a sample of the results from the first file, second file now drops, and now the results continue to drop. Third file, fourth file, and so on and so forth. So now we have a very simple setup to actually produce data and feed it to a streaming application. In the next segments, we'll actually see how to perform some operations against this data and also look at how different transformations result in aggregated data. We'll also look at how to run SQL against our dataset and how to perform stateful operations and keep the dataset in memory that we continue to update throughout every batch and different approaches of how we can do that more efficiently.

Refactoring Streaming Application: Demo

Now that we've created our first streaming application, let's refactor this a little bit so we can separate our business logic from the setup procedure. So let's go to our SparkUtils and define a getStreamingContext function. So def getStreamingContext. And then I want this to accept a SparkContext similar to the other examples, so SparkContext, and also a batchDuration of type Duration. Then we'll go ahead and create the function stub. Now this getStreamingContext function is going to be a little bit different than the previous getSqlContext and the getSparkContext functions. And the difference is that I want to be able to try and recreate these streaming contexts from a previously saved checkpoint if it's available. And if it's not available through the checkpoint, I want this function to be able to create a StreamingContext. However, I want that context to carry your Spark Streaming's application. So what I want to do is have this function accept another parameter called streamingApp, and the type of this parameter is actually a function, and this function takes a SparkContext and then a Duration and returns a StreamingContext. And I'll also hit Alt+1 to get rid of the left pane to get a little bit more real estate. So the plan here is to have getStreamingContext be a higher-order function in that it accepts another function called streamingApp, and this function takes the SparkContext and Duration as input parameters and returns a StreamingContext. So it's the responsibility of this streamingApp function to actually create a new StreamingContext. And it's the responsibility of our getStreamingContext function to either get the active StreamingContext if it exists or create one from the checkpointDirectory if a checkpoint actually exists there, and if those two cases fail, then to use the provided function streamingApp to create a new StreamingContext. So let's go ahead and implement the code to do that. So instead of recreating the code to the checkpointDirectory, we're going to pull that off the SparkContext. So we'll use sc. getCheckpointDir. And then we'll apply a match very similar to a switch statement. And you'll see that the getCheckpointDir method returns an Option of String. So our match is going to be case Some of checkpointDir. And so in cases where the checkpoint directory is actually defined on the SparkContext, we're going to want to create a StreamingContext. But instead of immediately trying to create a new context, we will try to get an active one, and we're doing this for the exact same reasons that we did it for the SparkContext where essentially you can recall this getStreamingContext function. Even if the StreamingContext was already created, it will return the active one without bailing out on you. And that means if you have functions that rely on having a SparkStreamingContext available, they can safely call getStreamingContext, and you won't have to necessarily constantly pass around the SparkStreamingContext. So we have this method for good practice. Now if I hit Ctrl+P to look at the signature of the method, it has two overrides, one that takes a checkpointPath, a creatingFunc, hadoopConf, and a createOnError Boolean variable, and the other takes a creatingFunc. Notice how this actually lends itself to what we're trying to achieve. However, the creatingFunc takes no parameters, so it's just a function that takes no parameters and returns a StreamingContext. So we're going to have to have a way to provide that. So knowing that our getStreamingContext function actually takes a SparkContext and a batchDuration, we can provide those to streamingApp and produce another function that takes no parameters and returns a StreamingContext. So we'll create a new function lateral val called creatingFunc. So we'll have its type as a function literal that takes no parameters and returns a StreamingContext. And we'll have the implementation again equal a function that takes no parameters. Again, this is an anonymous function in this case, which is going to be assigned to the creatingFunc literal. And so the anonymous function takes no parameters and actually calls the streamingApp function while passing in the SparkContext and batchDuration that we get from the getStreamingContext call. Now I only added the actual type here to try to explain this, but you really don't need this, and it can be a little bit confusing seeing the function and method call multiple times. So I'm just going to remove that. So we'll go back to our match statement. And now in case we actually have a checkpoint directory, we are going to provide the value for that checkpoint. I'll hit Ctrl+P to show you the method parameters. So the second parameter it's expecting is the creatingFunc. So we'll provide that. It also needs the hadoopConf, and we'll pull that off of the SparkContext. And then this is an interesting parameter. So what createOnError allows you to control is that if your application actually exits or fails and you try and restart it, and there's actually a checkpoint directory or a checkpoint defined that works with your application, and the getActiveOrCreate tries to recreate using that checkpoint, the createOnError controls what happens if the provided streaming application defers from the application code that was saved in the checkpoint directory. So let's say, for example, you make some changes to your streaming application, and then you restart your application, your code is now different than what was actually checkpointed. And currently Spark Streaming has no way to deal with that. So it's essentially asking you, What do I do in that scenario? So you can either tell it to go ahead and recreate the StreamingContext using your new code and basically disregard the checkpoint directory information, or just return a report failure. In our scenario, we're going to let it recreate the StreamingContext. Now we still have one more scenario to implement for completeness. So in case there was no checkpoint directory defined on the SparkContext, we're going to simply use the StreamingContext. getActiveOrCreate, again for safe measure, and then provide the creatingFunc. So this scenario handles when there's no checkpoint directory defined. And so now you'll see that the result of the match statement expression is a StreamingContext, so we'll define a variable called SparkStreamingContext (ssc), assign that to the result, and simply return that. So now we have code that uses the checkpoint directory if it exists. So the last thing for us to do here is actually set the checkpoint directory if it exists. So we'll get the checkpoint directory from the SparkContext, and that's an option, so we'll foreach on that. So for the checkpoint string, cp, we will use the SparkStreamingContext (ssc) and set the checkpoint (cp). So to summarize, we examined the checkpoint directory from the SparkContext. If it existed, we tried to create the StreamingContext off of that. And if it didn't exist, then we tried to get the StreamingContext from the creatingFunc. And then we finally set the checkpoint directory on the StreamingContext that was created for the new execution. So now we can go back to our StreamingJob and have it use our new getStreamingContext function. So what I want to do is actually move all of this inside of a new function. So we'll create a new def streamingApp, which takes the SparkContext and a batchDuration of type Duration. And let's go ahead and Alt+Enter here for the auto-imports to work its magic. So, remember, it's still the responsibility of this function to create a new StreamingContext as so whenever it's actually called upon. So if I go back to SparkUtils, you'll see that we will either try to get the active context if it exists or retrieve it from the checkpoint directory or call the creatingFunc to get a new StreamingContext. So it's still the job of the creatingFunc here to create a new StreamingContext if it gets called upon. So streamingApp should return towards the end the StreamingContext to the caller. So now inside of the main function, I have a new function called streamingApp, which we can now use to provide the StreamingContext. So now we'll create a new variable called SparkStreamingContext (ssc), but we'll use the new getStreamingContext function that we just created, provide streamingApp as the first parameter, the SparkContext (sc) and the batchDuration as the following parameters. So now that returns a SparkStreamingContext that we can then use to launch our program. So why did we go through all this? The first thing is that it really allows us to encapsulate our business logic in a nice function. And by doing so, we can now compose function calls together. So if we had multiple functions that we knew did specific things with a StreamingContext, then we can actually call them one after the other and not worry about the setup and teardown and making sure that we're using proper checkpoints and what have you. So our focus now can be solely on our input DStream and implementing our application's business logic.

Spark Streaming with SparkSQL Aggregations: Demo

In the previous clip, we were able to separate the business logic from our Spark Streaming setup. So now that we have that taken care of, let's go ahead and start creating aggregations in a streaming fashion. So we'll keep our textFileStream as our input for now. And we're going to apply aggregation similar to what we've previously done in our BatchJob. And for now as we're still explaining concepts, we're not going to complicate things and start doing code reuse. So what we're going to do is to grab some of the code that we've previously created in the BatchJob and then see how we can apply it in a streaming fashion. So in our BatchJob, we started with an inputRDD of type RDD of String, and similarly in our StreamingJob, we now have a DStream of String. So let's try and perform a similar aggregation that we did in our BatchJob. So all I'm going to do is to copy all of the code starting from the flatMap where we create an RDD and transform that to a DF of type Activity and continue all the way to include the activityByProduct aggregation. So back to our StreamingJob, remove this print statement, and I'll just paste what we have here. So there are obviously a lot of problems, and IntelliJ starts to highlight almost every line in red. But the good thing here is that we can use Spark's transform and foreach operations on the DStream to drop down to the RDD and not have to recode this at the DStream level. And an obvious benefit of doing it this way even if there are potentially equivalent transformations that you could use directly on the DStream using the DStream API is that you get code reuse. So eventually the same code you use in a DStream transform or foreach function can be reused in your batch layer and will actually extract these bits of code at a later point into a concrete function that we'll call from both batch and streaming instead of the lambdas we're using here. So we'll go ahead and take the textDStream, and then apply transform. You'll notice that the first step in the previous batch operation was to take the inputRDD, apply a flatMap with some transformations, and then return a result. So we're not syncing the records just yet. So transform works here well. And I'll hit Ctrl+P here to see the overrides. The transformFunc has two overrides, one that you can use where the DStream actually passes in the Time alongside with the RDD, and the other is simply the RDD, and that's what we're going to be using for now. So we're going to need a function that takes an RDD input of type String and returns an RDD of type U. So we'll call this RDD input. Now we can simply copy this code over and move it inside. Now we'll have to get rid of the toDF because the transform expects an RDD return type, not a DataFrame. It doesn't work with DataFrames like that. So what have we done here? We simply took the DStream of Strings, applied transform, so we dropped down to the RDD level, received an input RDD of type String, and then we were able to reuse the exact same code that we wrote in the BatchJob. So if I were to look at the type of this entire expression, you should expect to see a DStream of Activity, and that's exactly what we get back. So let's assign this to a variable called activityStream. So for the next bit, I'm actually not going to use any of this. I'm not going to do any timestamp transformations, and I'm not going to calculate the unique visitorsByProduct. We're just going to focus on activityByProduct for now. And we will actually get back to visitorsByProduct at a later point after we discuss stateful operations. So how can we work with activityByProduct and run this same SQL statement reusing the same code? Again, very similar to what we did earlier, we can take the activity DStream, so the activityStream, and apply the transformFunc. And, again, this takes an RDD. And this time the RDD is of type Activity, and we'll have to provide some transformation here. So now for us to be able to use the same code that we have here, we need a SQLContext. So let's go ahead and define our SQLContext. So val sqlContext = getSQLContext, again the function that we created, and we'll provide it with the SparkContext. We also need to import implicits, so we'll import sqlContext. implicits. And the implicits under sqlContext are what allow us to transform an RDD to a DataFrame and also provide a few other useful implicit conversions. So having said that, let's create our DataFrame out of the RDD, and then let's register the TempTable that this SQL statement uses called activity, so df. registerTempTable("activity"). And you'll notice that IntelliJ complains about the transform function because we still haven't returned an RDD. All it's saying is it can't resolve the reference transform with the signature that it sees. It's actually expecting that the function returns an RDD of type U, and we still haven't returned anything that satisfies that type. So let's go ahead and copy this code here but without the cache because we don't need it now. We still don't have any evidence that we actually need that. So now what we've been able to do is take a DStream and then write simple SQL statements against it. And the SQL statement will run against each RDD. And, remember, an RDD represents a batch so it will run for every batch defined by the batch interval, which is 4 seconds in our example. Now the transformFunc still shows a lot of ugly red because we still haven't returned an RDD of type U that satisfies the function. So now we have an activityByProduct DataFrame that is the result of this SQL statement. So let's take that DataFrame and map against every record. And we'll call each record r. And so r goes to, and we need a return type. Now the data type of r is a row class, so it's an instance of a row object. And for the return type of this map operation, I want to return a key value pair. And the reason for that will be apparent when we start discussing stateful operations. But essentially I want to return a tuple of two where the first element is the key and the second element is the value. And the first element is actually a composite key, so, again, another tuple. And what I want to return here is the product and timestamp\_hour as the key and then everything as the value. So we'll use a row object r. In this scenario, we're just going to access them by their index and type. So r. getString. And the index is 0 base, so I want the first column, which has an index of 0, and that returns the product because it is the first column back from our SQL statement. And then I also want to get the timestamp\_hour. And that's actually a Long. So now this is now a tuple of two items, which actually represents the key in our larger tuple that includes everything. So that's the key. Now I could go ahead and just return another tuple of multiple items, but that's a little bit vague working with it after you get out of this function. You'll have multiple tuples of tuples, and it gets really messy. So we're going to go ahead and create a new domain class called activityByProduct. So we'll go to our domain package, and then add a new case class called activityByProduct, which has a product of String, timestamp\_hour, purchase\_count, add\_to\_cart\_count, and page\_view\_count. Now back to our StreamingJob, now we can actually return an activityByProduct and then Alt+Enter to do the proper import. And then I can simply do an r. getString for all of the values. So now after mapping the DataFrame, the return type of this map operation is now an RDD of a tuple. And the tuple has two items. The first is a composite key representing the product and timestamp\_hour of type String and Long. And the second is our activityByProduct case class. And now by doing that, that actually satisfies the transform signature. And so the result of this entire transformation if I do an Alt+= on that, it returns a DStream of the last operation that we did. So now that we've done a couple of transformations and aggregations, all we have left is to define an action for this to kick off the job. So I can simply do a. print towards the end, and that will act as our sync or output operation or action, whatever you want to call it, that will cause the DAC to execute. Now I could run this and show you the results in the IDE, but it'll be more interesting to show you how this runs in Apache Zeppelin. And it will also be a nice introduction to show you how to set up zeppelin to run a Spark Streaming application.

Streaming Aggregations with Zeppelin: Demo

In a previous clip, we've seen how we can perform aggregations in a streaming fashion. In this demo, we're going to use Zeppelin to demonstrate running a Spark Streaming application similar to the one that we've built that performs aggregations on a streaming dataset. So I already have Zeppelin open with a notebook prepopulated with some of the code that we've already built. So you'll see that we have the case classes that we already have defined in our domain. And we're also importing the classes that we're going to use. Now I will highlight some of the differences between what we're doing here in Zeppelin and what we've done in our code in the IDE. The first thing you'll notice is that when we create the SparkStreamingContext, we have this @transient keyword. The transient keyword in Scala marks a field or value such that it doesn't serialize it in memory and transfer it over the network. In Spark terms what this really means is that when you try to access this object on a worker node, the object ends up being created on the worker nodes themselves instead of actually being created on the driver and the serialized and then passed along to the workers. So instead of the driver creating the object and passing it along to the workers, the workers themselves end up instantiating the object. And the reason we're doing this here specifically in Zeppelin for the SparkStreamingContext is that Zeppelin will try and serialize the SparkStreamingContext because it needs it in other locations. And to do that, it needs to grab the surrounding closure. And in Zeppelin's case, the surrounding closure's context is everything in the notebook. So that means that Zeppelin will eventually try to serialize everything just to pass around the SparkStreamingContext. This is a little bit more involved than I wanted to get into here, but just understand that this is a special case for Zeppelin that we have to create the SparkStreamingContext as a transient variable so that it doesn't attempt to serialize it along with everything else. If you don't do this this way, you'll actually get serialization exceptions. Moving on. So the next step is that we define the input path similar to what we did in our example in the IDE. Again, nothing really fancy here. It's the exact same code. We do the transformation on the DStream. And we grab the input RDD. And then we do perform a flatMap and return an Activity. So the result of this textDStream. transform is that we end up with a DStream of type Activity. And now that we have a DStream of Activity, we move on to transform again just so that we can drop down to the DataFrame and register a TempTable called activity and perform a simple SQL query against it. Now we could have actually done this SELECT statement and performed these aggregations on the DStream directly using reduceByKey and other actions and transformations that the DStream provides for us. But we're doing this to illustrate how you can drop down from a DStream to an RDD and then use familiar functionality. And this approach will definitely come in handy when we start sharing code between our batch operations and our streaming operations. So if you think about it, what we'll ultimately get to are functions that we'll call from both our batch code and our streaming code. Now for the purposes of this demo, I don't want to perform this map here. And I want to actually register a TempTable, and let's call that TempTable activityByProduct. And you'll probably notice that I've already accounted for the fact that we're going to make this change in the SQL statement we have below. So now that I'm actually not returning anything and merely just registering a TempTable at the end of this transform, this really shouldn't be a transform. And all we're doing here is a foreachRDD. Remember that your transform expects that you're returning something at the end, and in this case, we're not, we're simply registering a TempTable that we're going to use in a subsequent query. And then we're also going to remove this print statement. So the idea here is that we're going to run very similar code. However, we are going to run a foreachRDD, so essentially a foreach for every batch, and then run the SQL statement. And the result of that SQL statement will register a TempTable called activityByProduct, which we should be able to query in a subsequent step. Now the SELECT statement is really very simple. It's just selecting all the columns with one simple difference. We're actually selecting the timestamp\_hour in minutes instead of hours just for display purposes. And also you'll notice up top that we're not dividing by the number of milliseconds in an hour. In fact right now, we're dividing by the number of milliseconds in a minute. So the granularity of our dataset is minutes just so we can actually show you something and not have to wait for hours to present a meaningful demo. Now we're almost ready to run this example, but one last note in that how you can stop your SparkStreamingContext without affecting the status of your Zeppelin notebook. And the way to do this correctly is to use the StreamingContext. getActive, which returns a list of the active StreamingContext, essentially in this case only 1, and then loop against those with foreach, essentially every StreamingContext will be represented by this placeholder underscore. And then we'll call stop on each StreamingContext. However, we want to make sure that the Zeppelin SparkContext doesn't get stopped along with the StreamingContext. Remember, Zeppelin actually brings up its own SparkContext, so we don't want to stop that. And so we call this stop method with the stopSparkContext set to false and stopGracefully set to true. And then, finally, you can choose to use a ssc. awaitTerminationOrTimeout. If you have a print statement, that will cause Zeppelin to constantly run through that print statement and return the results based on the batch interval. Of if you had a specific window based on that window length. And we'll actually look at windowing in the next clip. So for now, I'm actually going to go ahead and disable these two from running, so I'll disable this one from running, and I'll disable this one from running as well. So now we'll go ahead and ask Zeppelin to run through the entire notebook. So you see that it defined the Activity class and ActivityByProduct class. It also defined our SparkStreamingContext and then defined our textDStream and activityStream. And then you should actually see Zeppelin complain about the activityByProduct table that it's not found. And the reason is that there's no data, so there aren't any RDDs to iterate over and actually create the table. So this is interesting in that the activityByProduct table hasn't formed yet. And, hopefully, with your understanding now of DStreams are formed, you can kind of make sense of what's happening here. A DStream is an abstraction that is created by stringing together multiple RDDs. And each RDD within that DStream represents the data received within that batch interval. So if no data is received at all, then there are no partitions that are being created to formulate this RDD. And as a result, there are no RDDs to foreach through in this example. And so that's why the table doesn't exist yet. So why don't we make this a little bit more interesting and actually start producing some data. So I'm going to bring the IntelliJ IDE just side by side here. It is really small here, but all I really need is the LogProducer and just right-click on that and run the LogProducer. And then I also have the input directory here for us to look at. And so I'll continue hitting Shift+Enter to see the results of this query. And it should start executing and returning results once there's data. I'll actually change this to a line chart. So let's just run this again to see if we get data now that there is data being produced in the directory. And, indeed, we are seeing some data coming through. Let's actually go to Settings and add the page\_view\_count and the add\_to\_cart\_count. So now I have the purchase, page\_view\_count, and add\_to\_cart\_count. And let's continue to run this while the data is being produced. So, again, no data available for this instance. There wasn't any data. Notice that it's just the batch that we're looking at that may or may not have data based on the data being produced and the time it hits that batch. And then I can keep running this, and you'll see the data constantly change. Now also notice that every time I run the query, it's returning a completely different set of results. So the set of results is completely constrained to the batch that we're looking at. What if I wanted to run the query against a larger period of time? So my batch period is currently set to 4 seconds. What if I actually wanted to run it for 16 seconds or maybe 24 or maybe a couple of minutes? Well that's where windowing in Spark comes into play. And that's exactly what we're going to look at in the next segment. And there's also another extension to this idea. So what if you not only want to look at the data in a larger period of time, but you also wanted to keep track of every dataset that comes through, and you wanted to keep track of some state in memory that you constantly accumulated? So instead of running through a batch or potentially a window and discarding the result after that window or batch are complete, you want to actually maintain a state that you keep around and constantly update with every batch or every window that comes through. And that's exactly what Spark provides with its stateful functions, and we'll look at those immediately after we talk about windowing. So let's go ahead and stop our SparkStreamingContext here. Don't forget to do that. Otherwise if you try and run this again, it will fail. And I'll need to re-enable this to make that happen. So Enable Run and then Shift+Enter to stop the StreamingContext.

Summary

In this module, we saw how the Spark Streaming Context was the center of the universe for a streaming application and how it uses the SparkContext. We went ahead and created our streaming application with streaming aggregations by dropping down to the RDD and using Spark SQL with transform and foreachRDD functions. We also covered the streaming receiver model and the requirements a receiver-based approach has on processor cores. And we'll see how that can change when we look at streaming data from Kafka in module 5. We also briefly discussed checkpoints and how they're needed for certain scenarios of a streaming application. We'll actually start off the next module with the further discussion on checkpointing, and we'll continue that discussion as we look at different approaches to receive data from Kafka and requirements for guaranteed delivery. In the next module, we'll continue to build on our streaming application right where we left off and cover window operations, stateful transformations, and we'll re-introduce the unique visitorsByProduct calculation as we discuss how to estimate cardinality or a uniques calculation in a streaming fashion, which is at the heart of being able to use streaming applications without losing speed and flexibility. So now that we have a foundation, we'll also start setting ourselves up for code reuse in the next module. See you there!

Advanced Streaming Operations

Intro

Hi! My name is Ahmad Alkilani, and welcome to this module on Advanced Streaming Operations with Apache Spark. In this module, we'll introduce more interesting things we can do with Spark Streaming and a combination of other libraries. We'll start off with a proper introduction to checkpointing in Spark. Then we'll introduce window operations where you can start to use Spark Streaming to run calculations in sliding or tumbling windows. And we'll look into answering one of the questions we raised in the previous module in that how can we use stateful transformations to store and record some state that you wish to keep and maintain throughout streaming batches, and obviously different ways of how we can act on that state and use it. And we'll end this module with a look at how we can re-introduce the unique\_visitors calculation and calculate cardinality in a streaming fashion while also not having to store every source record to get a good estimate for uniqueness. The trick here is to use a technique where we wouldn't have to strain memory and system resources by keeping every single source record. This module really starts putting what we've learned to practice, so let's get started.

Checkpointing in Spark

Checkpointing in Spark is a feature that can be used in normal non-streaming Spark applications if the execution graph is large enough to merit checkpointing in RDD. This serves to store its state so that it's lineage need not be stored entirely in memory. However, checkpointing is generally used or even required with certain types of transformations in streaming applications. There are two types of checkpointing operations. The first is metadata checkpointing. This is basically persisting configuration, DStream operations, and information about incomplete batches that have yet to be processed. Don't confuse this with the ability to recover from received data that has not yet been processed. The reliability and recoverability of received data depends on the receiving mode used and whether or not supporting characteristics are in place like a write-ahead log or for reliable receivers used, and we'll discuss the different modes of receiving data in the next module when we incorporate Apache Kafka. For now, know that this mode of checkpointing specifically targets recovery from driver failures. If the driver fails and you don't have checkpointing enabled, then the entire DAG of DStream execution is lost in addition to the understanding of state for executors. So metadata checkpointing helps Spark applications tolerate driver failures. Keep in mind that a driver failure actually also means that you lose your executors, so restarting the driver is kind of like restarting your application from scratch except that we use GET or CREATE on the Spark and streaming contexts so the driver will attempt to recover the information it needs from a checkpoint and relaunch executors in their previous state. The second type of checkpointing is called data checkpointing, and this is useful for stateful transformations where data needs to be stored across batches. Window transformations and stateful transformations like updateStateByKey and mapWithState, as we'll see shortly, require this. Now you can checkpoint RDDs on your own, but simply using these transformations and enabling checkpointing on the StreamingContext as we've done already in previous modules essentially takes care of all we need to enable both metadata and data checkpointing alike. So now you have an idea of where and how checkpointing is used, and we already have everything set up properly to use it. Let's go ahead and take a look at window operations next.

Window Operations

To explain window operations in Spark Streaming, let's start with something that should look familiar to you. And so this is an excerpt from the slide where we explained how DStreams are formulated from RDDs that rely on the batch interval. And as we know, the RDD that represents each batch interval consists of partitions that fill up with data as it streams through into the Spark universe and your application. So let's take this picture and abstract it a little bit and represent these RDDs with these small blocks. And to simplify this example, we're going to assume that these RDDs hold an integer value, so the value inside of these blocks, 6 and 3, just represent the value that these RDDs contain just to simplify this example. So in a similar fashion, if we were tracking through an entire timeline, we would have a lot of RDDs that we're tracking the progress of. Also, let's assume that the batch interval is 2 seconds. So each RDD represents 2 seconds' worth of data. So to create a window of a stream of data in Spark, you have to define two things. The first is the window size. And so in this example, we're defining a window size of 6 seconds. And the second is the slide interval. And the slide interval controls two things. It obviously controls the interval by which the window will slide, so in this example if I had my slide interval defined as 2 seconds, the first slide would drop the RDD with the value of 7 and then move on to include 3, and then the second slide would drop the RDD with 1 and move on to include the RDD with 2. The second thing it defines, which is not too obvious here, is that it defines the interval by which your operations are going to actually evaluate. So if you have a slide interval of 2 seconds, then you're evaluating your window, your entire window is getting evaluated every 2 seconds. If you had a sliding window of 30 seconds on the other hand, then your window's actually going to get evaluated every 30 seconds. So the evaluation takes part every 30 seconds. However, the window size only contains data 6 seconds' worth. Obviously that means in that scenario that you're dropping data on the floor, but that's essentially how it works. So in this example, you could also define the sliding window as 6 seconds, and that way your window's actually sliding the entire length of the window itself essentially achieving what looks like a tumbling window. So when you're performing a calculation of window of data in Spark, you get to define the window size and the interval by which it slides. And there is one restriction on the value of the window size and the slide interval, and that's that they both must be multiples of the batch interval. So if my batch interval is 2 seconds, the only values that you can use for the window size and the slide interval are multiples of 2. Similarly, if the batch interval is 1 second, then multiples of 1. And that's for obvious reasons in how Spark works. It's not going to attempt to split RDDs in half and figure out how to slide against that. It needs full RDDs, which are represented by the batch interval. So let's look at a more practical example. So, again, we're assuming the batch interval is 2 seconds and that represents each RDD, and then we're calling this the original DStream. And then we'll provide an indicator of a timeline to track how things evolve as we progress through the timeline. So let's start with a very simple window operation. So we'll assume we have a clicksDStream and these RDDs represent the number of clicks in a streaming fashion. And Spark provides a function called window where the first argument is the window size and the second argument is the slide interval. And by performing this window operation, all we're telling Spark is that we want to look at our data in a window of 6 seconds and whatever follows this transformation will execute every 2 seconds. So let's see how this works relative to our original DStream. So if the timeline moves forward, the first RDD that shows up will be included in our window because it's still within the 6 seconds. The second RDD will get included just as well. Similarly, the third RDD. And so that's our 6-second window size. Now typically when you perform window calculations, you actually have an idea of what type of operations you want to perform. And so I could, for example, perform a clicksDSstream. window and then follow that by a reduceByKey operation for example. But Spark provides some convenience methods, for example, the reduceByWindow operation. And while this acts as a convenience method to replace two operations, a window and then a reduce, it actually has some performance advantages as we'll see in a minute. So let's look at what happens if we use this reduceByWindow. So the reduceByWindow operation takes an input function, and the input function expects two fields of a certain type and expects you to return a single value of that same type. In our example, this type would be integer, so a and b here would be integers, and then it expects you to return a result. So we're essentially performing a reduction. And a in this scenario would represent everything the window had before introducing a new RDD into the window. So before we introduce the RDD with the value 6, the a value would have been the sum of 7 and 1. So a would be 8, and then the reduce operation would perform the 8 + 6. So let's actually represent this by a windowed DStream timeline. And so at the current time t, you would get a single RDD with the result 14 resulting from the reduction of the RDDs that you have in the window. Now let's say our window slides another 2 seconds, so we'll slide our window. We drop the first RDD, the window slides, the new RDD 3 joins the window, and then we perform the calculation again, returning your result of 10. And so if we continue with this analogy if we want to slide the window again, then we will drop 1, and we will perform the slide, 2 will join the window, perform the calculation, and we'll get the value 11. Now as I mentioned earlier, these special window operations like reduceByWindow and reduceByKeyAndWindow have another method signature that is aimed for optimization. So far what Spark needed to do to perform these calculations is to actually keep track and keep a hold of all of the RDDs that compose the values in your window. So if your window spans a couple of minutes and you have a lot of RDDs in that window, the Spark would actually have to actually keep track and keep a hold of all of the RDDs that compose your window. And that's actually why when you're performing window operations, Spark Streaming requires that you have checkpointing enabled because it needs to checkpoint this state somewhere or else it'll lose this information. It can obviously recalculate it, but it's better to just retrieve it because it becomes expensive. Now the optimization is that if your operation actually has an inverse operation that you can perform, for example, in the plus scenario, it's an operation that has an inverse. So we can actually use another method signature for reduceByWindow which provides that inverse operation. So now we have an operation that introduces new RDDs or batches into the window. And we also have an operation that is responsible for removing old batches from the window. So let's actually see how this works. So let's bring back the previous RDD, which is 1, and the previous value that we had calculated. So the general idea here now that you have an inverse function is that Spark really doesn't need all of the intermediate information, so it doesn't need the 6 and 3, and all it needs is the 1 coming from the old operation, and the previous value that we've calculated, which is 10. So we'll get rid of 6 and 3, and we'll see how this works. So let's map the fields of the anonymous functions that the reduceByWindow operation takes. So a is represented by the previous value 10, and b represents the new RDD that we're just about to add to our window, so that's the value 2. And so if you perform the reduction operation, a + b = 12. And then now 12 represents the value x in the inverse operation, and y represents the RDD or list of RDDs that you're about to slide out of the window. So in this case, that's the RDD with the value 1. And so if we subtract 1 from 12, we end up with the correct result 11. And so the idea here is that if you have an operation that has an inverse, then you should really take advantage of this. You'll save yourself some memory and processing time.

Visualizing Stateful Transformations

In the last demo, we presented two problems. The first question was around how to decouple operations from the batch interval, and we now know how to do that with window functions on DStreams where we define a window and a slide interval. The second question was around our ability to maintain some state throughout the entire application's life based on data received from a DStream. And Spark also provides means of maintaining state using either the updateStateByKey or mapWithState functions. We'll go through an in-depth look at the functions Spark provides to map state, but it's helpful to visualize what we mean by tracking state so you have a better handle on the code and demos that follow. You can think of a stateful transformation in Spark as some sort of global bag of information or global state that you have available to you to store information on the side. In reality, it's a distributed bag of information so the physical implementation isn't restricted to what one machine can handle. This bag of information contains the global state that you want to track and has a type. A major difference between using statement transformations and a reduce or aggregation function, for example, is that the state can have a completely different type than the data it receives. It's really up to you the implementer of the state function to define what it is that you want to keep in here. Furthermore, the state is global, meaning it's kept beyond the scope of the batch or window operation. Let's take a practical example. Let's say we're doing a car study, and we're tracking car types as they pass by on a high traffic freeway. The ask is that for each car type that passes by, we want to add a new field that tells us how many cars of that same type have already passed by. So this tells me a few things about what we need to track in our state. First, we're tracking state by key, and that key is the car type. And for each car type, we're tracking a running sum, so our state is actually a numeric value. And we'll use an integer for that. So our global state now has a type, and the type is integer. That's the state we're tracking per key, and the key is a string that represents the car type. And I'm using an integer here just as an example, but your state could just as easily have been a complex type, and it doesn't need to be something that is associative. The implementation of the state function is entirely up to you. So on our little example here as we process data from different types of cars, we'll continue to update the state for each car type. That sounds good. Now how do we add information from this state back into our stream? Well Spark does this a little differently depending on the implementation, and we'll see how that works in more detail, but this should give you an idea about the components involved in a stateful transformation. There is your original stream of data, in this case the cars on the freeway. Then there's the state itself, which always has a type that you define and is tracked by some key. There's the state function, which is something you implement, tell it how to take data from a stream for each key and translate that to the states you're tracking by also providing you with the previous value from the state. And, finally, there's what do you want to return as a result of the state transformation that becomes part of your original stream? Now that we have these pieces in our minds, let's take a look at the functions Spark provides for stateful transformations.

Stateful Transformations: updateStateByKey

So as mentioned earlier, Spark provides two stateful transformation functions. One is called updateStateByKey, and the other is called mapWithState. updateStateByKey was the only implementation Spark provided for quite some time, and mapWithState was introduced with Spark 1. 6 to address some of the issues updateStateByKey suffered from specifically around ease of use and performance. We'll examine both of these in the move to our code to see how we can use them. Both of these functions come with a slew of overrides, so we'll examine the most common parameter lists starting with the updateStateByKey. I'm going to bring up a segment of the code where we last left it in a previous demo. If you recall as we used transform on the activityStream DStream and performed some aggregations, we ended up with a map that created a key value DStream where the key was the combination of the produce and timestamp\_hour and the value was an instance of the case class ActivityByProduct. You may also recall that I mentioned we will clear up why we are doing this in a later section. Well that was specifically to enable tracking state by the product. The stateful DStream's functions run under the assumption that you are always tracking state by some key. What that key is obviously depends on your use case. In our example, we're going to accumulate activity for a product, so the key is the product. And we're also interested in maintaining that state for every product and hour combination. So not only a product but also the source system's timestamp. Spark Streaming as it is today doesn't behave like a complex event processing engine so the source system timestamp has no impact on the streaming window at all, and Spark doesn't really care about it. It's simply part of your data. So to accommodate something like maintaining state based on the source system's time, we can do something like this and use a composite key to key off based on the natural key and a timestamp. So let's focus on the last bit of this code here and see how we can maintain state with the data presented. We'll start with updateStateByKey, and let's try and visualize the state that we're maintaining for each key as a table. So we'll have the key and the state, and as an example of a key, let's just call it A. And then let's say that our state is just a tuple of three values. And these values are of the integer data type, so the state is a tuple of three integers. And as the state is maintained per key, let's just add an additional key for illustration. Now remember our code we started with a DStream, and we performed some aggregations on it. And then we finally performed a map to return a keyed DStream where the key was the product and timestamp and the value was the ActivityByProduct. So how do we go about maintaining state per produce and timestamp where the value is the ActivityByProduct? So in this example, what I really want our state to be is the purchase\_count, add\_to\_cart\_count, and the page\_view\_count, and that's represented by this tuple of three integers or tuple of three Longs when we actually go and work with the code. So in calling the updateStateByKey function, it takes a type of the state that you're actually maintaining. So in our case, the state is the tuple of three integers. And updateStateByKey takes a function as one of its parameters that is expected to return an option of that type, so an option of the state type. And the reason it returns an option is that you can opt to return either Some of that state type value or None. And by returning none, that essentially removes that value from the state. So, for example, if you had a key and under a certain condition, you wanted to remove a value from its state, then you can just return none, and that key's state would drop out. So now this function that it takes that should return this option of the state type also takes two parameters. The first parameter is a list of the new items per key. So the updateStateByKey function actually gets called for every key. So imagine for the key A, the updateStateByKey function gets called, and a sequence of all of the values for the key A would be provided to your function. And then the second parameter that it takes is that updateStateByKey provides you with the current state. So whatever the value of the previously maintained state for that key is also provided as an option. And, again, it's an option because there might be a state for any specific key. It may be the first time that we see this key or maybe you saw it earlier and you decided to remove it. And then as part of processing new data, you get this key again. So there's a possibility that the previous state actually doesn't exist. So you either get a sum of the previous state type or a none indicating that there was no previous state. And that's certainly the case for the first RDD of the DStream that gets processed. So to reiterate, the updateStateByKey function takes another function, so it's a higher-order function, and the function it takes is expected to take two parameters. The first is a sequence of all of the new values for the key, and the second is the current state for that key. And then your function that you provide is expected to return an option of the state that you're trying to maintain. And in our case, we're trying to maintain a state of the purchase\_count, add\_to\_cart\_count, and page\_view count per key. Now I'm not actually showing the implementation of this function. All we're showing here is that it returns an option of the state type, which is a tuple of three ints, and the parameter list it expects. So we mentioned that you get a sequence of the values for every key. But it's not necessarily every key that the current RDD that you're processing as part of your DStream actually has. So as part of the DStream, remember, it contains a list of RDDs, and you're processing one RDD at a time in a micro-batch type fashion. So the RDD that you're working on may contain a list of five keys. However, your state may have a list of 1000 keys that it's already accumulated as part of running through your RDDs one by one and running through your dataset. Because the updateStateByKey function needs to give you the option to remove or drop out one of the keys from your state, it actually iterates every time over the entire state. So in this example, if your RDD in that batch only had the keys A and B, and your state had 1000 other additional keys, then the updateStateByKey function is not only iterating through this state for the keys A and B, but it's iterating through the state for every other key that your state maintains. So the 1000 keys that you already have in the state, updateStateByKey is getting called for them. Whether you do something with that or not is irrelevant. It's doing that because it needs to give you the opportunity to remove some keys from your state by returning none. So that means that the time this function takes is actually a function of the size of your state. So the bigger your state gets, the longer this updateStateByKey function takes to execute because it has to iterate over every key in your state that you're maintaining. And this performance problems is one of the chief complaints about the updateStateByKey function that gets addressed in the mapWithState function as we'll see shortly. So before we go ahead and visit mapWithState, let's apply what we learned here about the updateStateByKey function and go back to our code.

State Management Using updateStateByKey: Demo

So we're back in IntelliJ working on our project inside of the StreamingJob class. And we're looking at our activityStream DStream right where we're actually performing an aggregation. So we're going to change this print statement and replace it with an updateStateByKey. And as you can see, updateStateByKey has numerous overrides, and it's the first one that we're interested in. As you can see, it expects an updateFunc of type Seq of ActivityByProduct and an Option of the type S where S is the state that we're going to return. So let's go ahead and write our updateState function. So it's a function that takes parameters and returns a state. And the parameters are newItemsPerKey, which is of type Seq of ActivityByProduct. And then we also get passed the currentState. And, remember, the currentState is an Option because it may not exist yet. There may not be a currentState for that key yet. And it's an Option of the state type. So in this case, we're returning a tuple of three items, which represent the purchase\_count, the add\_to\_cart\_count, and the page\_view\_count. So a tuple of three items of types Long, Long, and Long. So let's go ahead and implement our function. So the first thing we want to do is get the value from the currentState. So we're going to do a getOrElse so we can actually return something if the state doesn't exist. So we'll return 0, 0, and 0. And we'll assign this to three variables, so var purchase\_count, add\_to\_cart\_count, and page\_view\_count. So now these variables either carry the values of the previously saved purchase\_count, add\_to\_cart\_count, and page\_view\_count in the previous state or the value 0 because the previous state or what we're calling the currentState actually doesn't exist. So now what I want to do is to iterate over the new items, so newItemsPerKey, and then foreach on that. So foreach ActivityByProduct a, we are going to perform an action. So we'll take the purchase\_count from our state and add to it the new value a. purchase\_count. Similarly, we'll take add\_to\_cart\_count and also add to it a. add\_to\_cart\_count. And we'll do the same thing for page\_view\_count. So what we did so far is that we grabbed the values off of our currentState, saved it into three variables, and then we iterate over the sequence of ActivityByProduct for each key that we received and add the values of each purchase\_count, add\_to\_cart\_count, and page\_view\_count to the variables that we've created. So now the only thing that's left is to return the state. As it is right now, I don't have any conditions by which I might deem the state unnecessary, so I'm just going to return Some and then a tuple of the three values that we've accumulated. And there we go. So now we take activityStream, we apply a transform to it, perform an aggregation, and then save the results of that into a state that we maintain. So let's give this new DStream a name, and let's call it val statefulActivityByProduct. And then if I Alt+= on this, you'll see the type, and you'll see the result of the updateStateByKey transformation is that it returns another DStream of the same key type, which is String and Long. So that's our product and timestamp. And now the value is no longer the ActivityByProduct that we had earlier and instead is replaced by the type of the state, which is a tuple of three Longs. And then finally notice that we still don't have an action, so we'll take the statefulActivityByProduct DStream and then simply perform a print. And let's say we return the top ten results. So this looks good. However, we still have one problem to solve. Our state constantly gets bigger depending on the number of keys that we receive and obviously the time that our streaming application remains on line and up and running. So we want to be able to do something that removes keys from the state if they've been in the state for too long. And too long is obviously relative depending on the type of problem that you're trying to solve. So let's see how we can do that. So we'll start off by checking if we actually received any new items for the key that we're working on. So if the new items sequence is empty, then we want to do something. So what we want to do here is that if we haven't received a new value for a particular key, we want to check if that key has remained in our state for too long. So we'll use the current timestamp, so System. currentTimeMillis. And so we need to capture the last time that a state for this key has been modified. And to do that, we actually have to capture that timestamp as part of the state that we're saving. So we'll go back into the return type for our state and add one more variable, which is the System. currentTimeMillis. And we'll actually modify the return type of the currentState. So we have one more Long in the list. And then also we need to modify the getOrElse with a default. And, again, that will be the System. currentTimeMillis. And I'll remove the left pane to get more real estate. And so what's left is that we have to assign this to a variable, and so let's call that variable prevTimestamp. So what we've done so far is that every time we update the state with a value, which is Some, we actually return the currentTimeMillis along with the actual values that we care about. And then also when we grab the state for each key, if there is no previous state, then we assume the currentTimeMillis as the time that this state was initiated for that key. And then let's also create a variable called result that we'll use to hold our state value, so it's return type is Option of type tuple of four Longs. And we'll default it to null. So now back to our if statement. So if there are no new items for the current key, then we need to check if the currentTimeMillis minus the prevTimestamp is greater than some value. So let's say 30 seconds. But when we're doing something like this, we also have to keep in mind how Spark works in that it's actually a micro-batch system. And so we not only have to wait for the 30 seconds that we've determined here, but we also have to wait for the length of the batch interval, which was 4 in our case. And then, obviously, in a real scenario, you probably don't want to keep your state only for about 30 seconds. You may want to keep it for about an hour and, again, it's really dependent on what your use case entails. So if we determine that the state for this key has been in our state for too long, then we will return the results = None ultimately removing that key from our state. And then if that's not the case, then we simply mark result with the current state. So we make no changes because we actually didn't receive any new changes or any new items for this particular key. So we'll just return Some and then the tuple of values that we already had in this state for this key. And in the case that there was no state, then we've already marked them with a default with the currentTimeMillis and 0s for the values. So what if we actually received some values for new items for this key? So else, and then we'll simply wrap our previous code here and then save this as the result, and then, finally, return the result as the state. So to summarize, if there are no new items for the key in hand, then we check the currentTimeMillis against the prevTimestamp from the key state, and if it's longer than a period that we've determined plus the batch interval, then we return None, essentially removing that key from the state. Otherwise, we simply return the state as is. And then if we actually receive some new items, then we'll iterate over those new items, add them to the state that we currently have, and then return that as the new state for this key. So how about we test this? So to run this, I'm actually going to do this in Zeppelin again. So I'm going to bring up Zeppelin. And then I have the exact same set of codes very similar to what we had earlier. And I changed it to match what we just did. So change this back to transform. And then we now have our map operation to get the key and the value and the updateStateByKey that we just put together. The only difference here is towards the end where I don't have the print statement, and I take the statefulActivityByProduct DStream and run a foreachRDD action on it, and I simply take this and return an ActivityByProduct case class again and register a TempTable. And the whole reason I'm doing this is simply to be able to visualize this through Zeppelin. So Zeppelin really needs a DataFrame or a table so that you can see those pretty charts. So, again, this is not something that we're going to add back to our code, but this is simply so we can look at the results. So let's go ahead and let Zeppelin run through this entire notebook. And then we'll go back to our IDE briefly, bring up the project, and run the LogProducer. And then while that's running, bring back Zeppelin, and also show the input directory so you now see that it's actually starting to produce it seems some new files. And if I run the SQL statement, you'll see that we're starting to show the results from our state. Notice how the state actually starts to accumulate instead of just completely start from scratch. And then also keep in mind that the LogProducer is random so the fact that we have a lot of purchases and barely any values for page\_views or add\_to\_carts is not really typical.

Stateful Transformations: mapWithState

We've seen how we can maintain state using updateStateByKey. So now let's take a look at a better approach for state management using mapWithState. So let's examine the function definition. mapWithState is a function that takes two generic types, the first called StateType, which is the type of the state you want to maintain, in our case this would be our Option of a tuple of three ints, and it also takes a MappedType, and I'll explain what that is in a second. mapWithState also takes a single parameter called spec, which is of type StateSpec of the key and value and a few other types we're not concerned with right now. This does seem more complicated when you look at it, so let's take a look at how it actually gets simpler than what we've used with updateStateByKey. When you call mapWithState, you supply a StateSpec, and the StateSpec works as a builder in that you continue to build on what the spec defines. The example here uses the required. function and a commonly used. timeout as well. So mapWithState takes a StateSpec, and the StateSpec takes a function that accepts a key, an optional value for that key, and the prior state associated with the key. And it finally returns an option of the MappedType. Now mapWithState function as its name implies actually allows you to perform two operations instead of just one. The first operation is updating the state, and you do that by calling an update method on the state that you've provided to your function. So your function gets a key, an optional value, and the previous state, and you simply update the state by calling the update method on the state field provided. In contrast to what we've seen with the updateStateByKey implementation, the return type of the function was actually what the state contained. In this scenario, when using mapWithState, you actually update a state variable instead. So that's the first operation. You obviously need to maintain a state. The second operation, which you can probably imply from the mapWithState name is actually very similar to a map operation on a DStream or RDD in that you can operate on the key and value and return anything you deem appropriate. Let's say, for example, in addition to maintaining state, you wanted to calculate some sort of score based on the input value. And you can perform your score calculation and simply return that as the final return type from your function. Remember, the state was already updated using the update function, and the return type is utilized for you to return anything you want. And that's what this MappedType actually is. So if we examine the return type of the mapWithStage function, it returns a mapWithState DStream, just an implementation of a DStream, with the key, the value, the state. Even though the state is being maintained, it also returns it as part of the mapWithState DStream. And in addition to that, you also get the MappedType, which represents whatever you decide to return from the function provided in the StateSpec. And in the case where you don't need to return anything, then that works as well, and your MappedType would be of type Unit. We'll work through a demo of this shortly so you'll see it in action. But before we do that, let's address this timeout. If you remember, there was a bit of code involved if we wanted to timeout keys and their associated state using updateStateByKey. We had to keep track of time and run through a bit of logic. And updateStateByKey is capable of retiring state because it visits every key in the state so you have the opportunity to remove records. mapWithState, on the other hand, doesn't visit every key in the state. So just by having that, it performs better. However, realizing that time-based state retirement is a common scenario, it comes with a simple approach where you can specify the timeout, and it takes care of retiring keys based on that timeout for you. It will end up visiting the entire state, however, only in increments of the timeout duration and not for every batch interval. So let's say you wanted to retire keys if they haven't seen an update every 30 minutes. Then you would supply that as the timeout duration, and a full state traversal will only happen when the timeout duration passes. A very important distinction between mapWithState and updateStateByKey is in what these transformations return. With updateStateByKey, the entire state is returned as accumulated by the execution of the state function against previous batches of RDDs. So with updateStateByKey, you get the full state back. Using mapWithState, on the other hand, you only get the portion of the state returned for the current batch or RDD, in addition to whatever arbitrary mapping you've also performed on that RDD. Again, it sounds a bit complicated so let's clarify. Let's bring in a familiar example. So, again, as we've seen in previous examples, each batch represents one of these RDDs, and the combination of these RDDs is abstracted as a DStream. So far, so good. Let's also bring up the accumulated state. Now let's assume we're processing the batch with RDD C. If I were to take the stream, let's call it myStream, run a mapWithState, and print the results, I'll get the keys, values, state type, and map type of what is returned by the current RDD and batch being processed. So in this case, I'll get the key C and its values, and the current state will now also contain the key C and its values. As we move on to the next batch with the next RDD, the current state will get D as a new key record. So now the current state contains C and D. However, the return stream from mapWithState only shows the current batch data. So this print statement won't show C and D in the current batch. It will only print the key D and its values. So the mapWithState DStream that results from the mapWithState transformation only returns data for the current batch unlike what updateStateByKey does where it returns data pertaining to the entire accumulated state. So mapWithState has a different approach to getting a snapshot of the entire state using a stateSnapshots transformation. You can apply stateSnapshots function on a mapWithState DStream, and it will return the entire state as opposed to only the current batch. So in our example after we process RDD with key D, the print that follows the stateSnapshots transformation will print a snapshot of the current state meaning it will print C and D similar to what the default behavior is for updateStateByKey. So let's go back to our code and see a practical example of how this all works.

Better State Management Using mapWithState: Demo

So we're back in IntelliJ. And now instead of using updateStateByKey, we want to use the mapWithState function. So we'll start this off with a. mapWithState and then Ctrl+P. You'll see the function definition actually takes a StateSpec with the type of the key, which is our tuple of String and Long. That's the product and the timestamp. And then the value type, which is ActivityByProduct, and then the StateType and MappedType as discussed. So what we'll need next is the StateSpec. Before we do that, let's go ahead and comment out the updateStateByKey transformation. So we'll start creating our StateSpec. We'll call it activityStateSpec, and we'll use the StateSpec builder, and we will need to provide it the StateSpec function. Now from these three overrides, we're interested in the third one, which takes a function that accepts the KeyType, an Optional value, and the State, and returns a MappedType. So what we need to do next is to provide this function. So let's go ahead and create a function called mapActivityStateFunc. And, again, the function definition was that it takes a key, and the key was a tuple of type String and Long. And then it also takes a value, which is of type Option of ActivityByProduct. And then it also takes the state, which is of type State of whatever the state type we want to provide. and then I'll hit Alt+Enter for IntelliJ to give me a list of classes to import that state. The one we want is the org. apache. spark. streaming. State. And then our state is a tuple of three Longs. So let's define the body of our function, and then we'll cheat a little here and copy what we've previously done and modify it. So the first thing here is that we don't need the previous timestamp because mapWithState will perform key expiry for us. And then the current state is simply called state. Now there's also still potential that the state for the current key that we're working on doesn't previously exist. So the state type actually has a getOption, which returns an option of the state type itself. If I do an Alt+= on that, you'll see it returns an option of our state type, which is the tuple of three Longs. And then we can simply do the getOrElse returning the types that we need. So far what I have is I've declared three variables--purchase\_count, add\_to\_cart\_count, and page\_view\_count-- and I've assigned them to the current state value for that key or defaulted them to all 0s. And given that the mapWithState takes care of the time, I really don't need any of this stuff. And also because we actually only get one value at a time, the value is simply an option of the ActivityByProduct and it's not a sequence of ActivityByProduct, then we don't need to loop against it. So we'll remove all of that. And then we'll clean up here as well. Now because the value is an option, we need to figure out if we were actually provided a value or not and then default to some value if we weren't. So we'll create a val called newVal, and it equals the actual value. We'll perform a simple match in case we're provided some value, so Some(a: ActivityByProduct). Then we'll simply extract the three values from a, so we'll return a. purchase\_count, a. add\_to\_cart\_count, and a. page\_view\_count. And then case default, which essentially means that we didn't get anything. Then we'll just return three 0s. So now newVal has either the value provided or defaults to 0s, and then we'll use newVal instead of a here to increment the variables that we have. So newVal, and the first one is the purchase\_count, copy this. And then the second one is the add\_to\_cart\_count. And then this third field in the tuple if the page\_view\_count. Now remember how we actually update state with the mapWithState function. We actually call state. update. So instead of returning something, what we're going to do here is call state. update and then provide the new state, which it now expects the tuple of three Longs. So we'll drop them right here. So that's all we needed to do here in our mapActivityStateFunc. However, we have the option of returning a mappedType. Remember that the mapActivityStateFunc can perform two operations. One is that we can update the state as we're doing here. And the other is that we can map against the values as if we're performing a simple map operation. So why don't we reintroduce our underexposed metric that we performed earlier in a previous module. So we'll create a val underexposed, and then we'll check if the purchase\_count is 0, then we'll simply return 0, else we will divide the page\_view\_count by the purchase\_count. And so we'll simply define the page\_view\_count divided by the purchase\_count as an underExposed product. And so we can simply return underexposed. So now we can go back to our activityStateSpec and provide our mapActivityStateFunc. And the main thing in the error that we're interested in is that it cannot resolve a reference for the mapActivityStateFunc with such signature. And the reason is that function here actually expects a function literal of the type function with the key value state and return type. So instead of defining a simple function here, we'll create a function literal which equals the function definition. And now we can simply provide mapActivityStateFunc to the function call here. So just one more thing to have a functional equivalent to the updateStateByKey that we've done earlier is to provide a timeout. So we'll say that the timeout is 30 seconds and then just align these. So now I can finally go back up to where we were calling mapWithState and provide it with the activityStateSpec. And you'll actually see IntelliJ still complain about this because it's a forward reference. So what we'll need to do is to take all of this code and then pull it up right before this, and all is well. So let's do a little bit of cleanup before we move on to the next section. So we actually don't need all of this commented code. So we'll remove it. And then how about we actually clean this up a little bit more. So instead of defining the mapWithState function here, let's actually move this out to a helper class. So we'll bring up the project, and then we'll create a new package, so New, Package. Let's just call this functions. And I'll also create a new package object. And then we'll take this out of here and move it into our functions\package object. And then simply go back to our StreamingJob, and all we need to do here now is to provide the proper import. So we'll import functions. \_. And then if we go back down, you'll see now that the mapActivityStateFunc resolves because we have the proper import. So this cleans up our code a bit, and it also gives us a nice structure for when we start sharing code between the StreamingJob and the BatchJob that we created in an earlier module. To summarize here, we created an activityStateSpec of type StateSpec. We used the builder function and timeout, provided the mapActivityStateFunc, and then we simply did a mapWithState providing that StateSpec. So now if I look at the statefulActivityByProduct data type, it's a mapWithState DStream of the type of the key, which is the String and Long, and the value, which is the ActivityByProduct, and then also the state, which is the tuple of three Longs, and then, finally, we have the type of the return type, which is the MappedType, which we simply use to define our underexposed calculation. We'll take a look at visitorsByProduct in the next clip and see how we can actually use sketch algorithms like HyperLogLog to keep a state of non-associative operations like counting the number of distinct visitors. As you know, a distinct operation can't be applied incrementally. So we'll introduce a data structure that we can use to perform a distinct operation in an approximate fashion. So in the next clip, we'll introduce a data structure that we can use called HyperLogLog that we can perform a distinct operation and return an approximation of the correct result.

Stateful Cardinality Estimation: Unique Counts Using HyperLogLog

In module 2 of this course, we examined two query use cases. The first was a simple aggregation for activityByProduct, and the second a slightly more complicated aggregation that calculated the unique visitorsByProduct. What makes the second calculation complex is that it's not something you can easily reduce to a single number and continue to iterate and add to. To calculate a unique count, you actually need the entire set for the entire period you're considering the calculation for to determine if any single observation is unique, and then if you expand the period, you can't simply take the result of a previous calculation and add it to the unique count from a new one. How would you know if any of the new observations isn't a duplicate of an old one? So you end up having to keep the entire result set and running unique counts on it again and again. This obviously has immense performance considerations, the number of records you have to keep, and, as a result, the memory used in addition to the processing required is in the order of the number of items. As our examples deal with web analytics and clickstream processing, you can imagine for a busy website this won't be easy to scale especially if the ask was to provide near real-time access to this information. This is precisely where sketch algorithms and estimators come into play. One of the most commonly used is called HyperLogLog, and it's used to estimate cardinality or unique observations. In comparison to a naïve approach, which can require 60 GB of memory to calculate unique observations for 3 billion events, HyperLogLog solves this with a high degree of accuracy with less than 100 KB. One of the things that makes approximations especially interesting when we talk about lambda architectures is the fact that you have this use case. There is a speed layer responsible for getting you answers quickly. And one of its traits is relaxing the accuracy a little bit in favor of processing speed and potential user error. The batch layer then comes in to calculate a more accurate view of the world at a later point in time. I'd even go as far as saying that the lambda architecture wouldn't be as useful to many if you didn't have an approach to solve problems like this. In applying HyperLogLog, we're actually going to be using a library from Twitter called Algebird that comes with an implementation of HyperLogLog. And if you're interested in a full walkthrough of how HyperLogLog and similar algorithms work in detail, feel free to check my Pluralsight course called Data Science and Hadoop Workflows at Scale with Scalding. Just to give you an idea of what HyperLogLog is all about, it's based on pattern observer rules. Its general approach is to take a hash of each item, and it uses the probability of hash collisions and pattern repetition to predict the cardinality of the input dataset. So that's generally what it's about. However, what's important for us to know in this course is that it's an associative in-memory data structure. That means it's a data structure represented by a class. In fact, it actually looks a lot like a two-dimensional matrix. And it's associative, meaning you can add multiple HyperLogLog instances, and that results in a new HyperLogLog instance. For the astute learner, you may realize that having something that is associative not only means that we can incrementally add to the value, but also means that we can distribute the aggregation across multiple nodes in a distributive fashion. So it not only saves us memory but also scales. So that's all we're going to cover in theory. Let's go ahead and see how we can reintroduce our unique VisitorsByProduct calculation using mapWithState and HyperLogLog.

Approximating Unique Visitors Using HLL: Demo

So we're back in IntelliJ, and for this demo, we're going to need the Algebird library. So go ahead and open your main project's POM file, and then make sure that you have the Algebird core 2. 11 dependency. Also for this demo, I went ahead and already baked in most of the code required because a lot of it is repetitive, and we don't waste time watching me type. So let's go ahead and take a look at what changed. So the first thing you'll notice is in the domain package, I added a new case class, VisitorsByProduct, which has the product and timestamp\_hour, in addition to the unique\_visitors count. And we'll be using this in a very similar fashion to how we're using the ActivityByProduct case class. And then also in the functions package, I've already included the mapVisitorsStateFunc that we're going to be using with the mapWithState transformation. We'll come back and revisit the details of this function shortly. But just note that it exists. So let's go ahead and open our StreamingJob. And then notice the import for com. twitter. algebird. HyperLogLogMonoid. And then if we scroll all the way down to this new section, so right after our previous section with the mapWithState for the activityStateSpec, you'll notice a new section that calculates the unique\_visitors by product. And then we also create a very similar StateSpec, similar to what we did with activityStateSpec. And we'll revisit this in a second. However, let's look at the details for the HyperLogLog implementation. So to work with HyperLogLog, the idea is that we're going to map over every item, every activityStream item, and then we're going to create an instance of a HyperLogLog for each visitor. So if I Alt+= on this, you'll see that it's of type HyperLogLog. Now to do that, we actually first have to define a HyperLogLogMonoid, and we're call it hll. You can think of this HyperLogLogMonoid object as a factory method. So you define the monoid with the parameters that we need, and in this case the parameter is simply the number of bits that the HyperLogLog is going to use. In this case, we're just giving it 12 bits, and that's more than sufficient for the amount of data that we're going to be running through our example. What's important here for you to understand that the size of the HyperLogLog determines how accurate it can get and the amount of potential unique observations it can handle. So we create an instance of the HyperLogLogMonoid, and then we simply use the activityStream, map against it, And for each activity record, we then produce a key, which is the product and timestamp\_hour and then use the HyperLogLogMonoid and simply provide it with an array of bytes that represent the visitor. So for every visitor, we will end up creating an instance of a HyperLogLog proper. So the result of this calculation is a DStream of the product, the timestamp\_hour as the key, and a HyperLogLog as the value. So we're going to mapWithState for the key and value. So then we provide the visitorSpec. So nothing new here. So here we use the StateSpec builder, provide the function and the timeout. So let's go ahead and examine the function. So the function here, again, we get a key, which is the String, the product, and Long timestamp\_hour, a value, which is an option of HyperLogLog if that key actually had a value, and then the prior state, which we're also saving as a HyperLogLog. So the first thing we're going to do is to check if the prior state actually had a value. Remember, this may be the first time we see the key, or maybe that specific key had expired. And so in case that state didn't exist, we will provide a HyperLogLog instance that has zero information. So the. zero on the HyperLogLogMonoid, you'll see that this actually returns an instance of HyperLogLog, which essentially carries an empty matrix. And so we store this state in the currentVisitor variable. And then we calculate the new state, which is simply looking at the current value, which is again a HyperLogLog but an option of HyperLogLog. So we perform a match. If the value is provided, then we add the current HyperLogLog to the new HyperLogLog. Again, HyperLogLog is associative, and it has a plus function defined on the class. So we can actually add two instances of a HyperLogLog together, and that returns a new HyperLogLog. And in case that value was not provided, then we just return the current state. So now that we have a new HyperLogLog that represents the current state added to the newVisitorHLL count, we provide that as our new state in the state. update function. So let's reexamine what happened here. We actually performed a map operation against every record that we're receiving from the activityStream. And for every record, we're creating a HyperLogLog instance for that visitor. So we end up having a HyperLogLog object that only accounts for that single visitor record. And then we continue to add this HyperLogLog object to the state that we're accumulating for the particular product and timespace\_hour. So for every product and timestamp\_hour, we accumulate an instance of a HyperLogLog object that constantly adds the current state HyperLogLog object to all of the new ones that are received. And we constantly update our state with the new HyperLogLog structure. Now as we know with the mapWithState operation, we may or may not choose to actually return another value. And in this case, we'll use the newVisitorHLL, ask for the aproximateSize. estimate, and that essentially returns a Long, and we just return that. So let's go back to our StreamingJob. So, again, to recap real quick, the monoid acts as a factory method, which we use to create a HyperLogLog instance for every visitor, which then we use the mapWithState transformation to accumulate a HyperLogLog state that we can then use to give us a cardinality estimate. And then we also provide a timeout here. So in our example, we're doing estimations for every hour per product, and we're trying to account for any source system lag or problems upstream by keeping the state available for an additional hour, so 2 hours in total. So now we have the statefulVisitorsByProduct DStream, which is a MapWithStateDStream with the key type which is the String and Long, the HyperLogLog for the state, and then the HyperLogLog for every value, in addition to our output which is the current cardinality estimate for this product and timestamp\_hour as the data is streaming through. I'm going to go ahead and press Alt+1 to get a little bit more real estate. So now I actually want to look at the snapshot of the entire state instead of looking at the state as and when each record passes through. So to do that, we take the statefulVisitorsByProduct and then ask for the stateSnapshot. And what that gives us is a DStream with the product and the HyperLogLog, essentially just the information within the state per key. So we can take this DStream, perform a foreachRDD so each RDD now is the key in HyperLogLog. We'll map against the RDD. We'll get a state record. And then we will create a VisitorsByProduct instance. This is the new case class that we've defined. And, again, each one of these records is a composite key, so it's a tuple of a tuple String and Long, so we get the product, we get the timestamp\_hour, and then this is the HyperLogLog, and we ask for the approximateSize. estimation. So the result of this rdd. map is an RDD of VisitorsByProduct. So we can very simply convert this to a DataFrame and then register a TempTable and call it VisitorsByProduct. Now the only real troublesome thing here is that although we're getting a stateSnapshot, we're actually getting that snapshot for every single batch. So for every batch that runs, we're actually asking for a snapshot of the entire state. And for that snapshot, we're going through it and registering this TempTable. Now there's nothing really wrong by doing that. But do realize that we are traversing the entire state for every batch interval. And that could get pretty expensive for multiple reasons. One is that if you're traversing the entire state and your state is really big, then you may end up wasting enough time that your processing time is longer than your batch interval time. So what that means is your application is now deemed unstable and will eventually fall over itself. And the second potential issue here is that let's say you're not simply registering a TempTable, and you're actually producing these records to an external datastore. Do you really need to update your external datastore with the information about your new state for every batch interval? You may have that requirement, and that's great if you do. This would work out just fine. However, if you don't have that requirement, and you can live with slightly slower updates to your external datastore, especially when you consider that this may be a large set of data, and you don't want to add too much load on your external systems. So what we can do is use windowing. Remember we learned how we can window a DStream and the advantages of a reduceByKeyAndWindow function with an inverse. So let's go ahead and give that a shot. So instead of immediately going for the foreachRDD, let's go ahead and try a reduceByKeyAndWindow. So you'll see this takes quite a few different options. The one we're interested in is the second one, which actually carries an inverse reduceFunc. And I'll go ahead and bring the slide from when we discussed this so you can try and piece everything together. So let's start off with the reduceFunc. So it takes two parameters, a and b. Each of these is a HyperLogLog. And it expects us to return a HyperLogLog as well. Now usually for a reduceFunc, what you would probably be doing here is just simply adding the two items a and b. However, as the result of the stateSnapshot is a little bit special, we have to do something a little bit different. Remember, for every batch, we're actually receiving a stateSnapshot. And the stateSnapshot is actually the entire state as it was updated as of this batch. So the last stateSnapshot is really all you really need. You don't need to add anything to it or you're actually going to introduce incorrect behavior and incorrect results. So if a were to represent the previous stateSnapshot and b were to represent the current stateSnapshot, then all we need is the most up-to-date stateSnapshot, and that's simply b. So all we return here is the value b. And then for the inverse function, again let's call this x and y, it takes two parameters, two HyperLogLogs, and then expects us to return a HyperLogLog. The reverse is true. So if x were the current stateSnapshot and y was an older version of the stateSnapshot, then all we need to do is return the current stateSnapshot. So what's left for us is to provide a slide duration, and let's say we want to slide every 30 seconds. Now remember that the slide duration has to be a multiple of the batch duration. So to do that, we'll simply divide by 4, which is our batch duration, and then multiply by 4 again giving us a solid integer that ends up being a multiple of the batch duration. So the results of this reduceByKey is that we only save or expose the snapshot every x seconds. And x in this case is 30, or in reality, it's actually 28 because that's the multiple of 4 that we're going to get. And so this reduceByKeyAndWindow will simply attempt to keep the latest value for the snapshot as it progresses. And then this foreachRDD will run every 28 seconds and in our case will produce a VisitorsByProduct, DataFrame, and register a TempTable. But in future modules, you'll see that we'll actually take this result and save it to Cassandra. So now that we've done all of this for the VisitorsByProduct aggregation, let's go ahead and revisit our ActivityByProduct and do the same thing. So in the last segment, we looked at our ActivityByProduct, we applied a mapWithState, but then we didn't do anything after that. So let's go ahead and bring in some code that does that. So we define an activityStateSnapshot, which is simply the statefulActivityByProduct, and we call the stateSnapshots on it. We simply do the very similar reduceByKeyAndWindow again keeping the latest snapshot state. And then we foreach on that and create an ActivityByProduct case class instance, toDF, and register that TempTable. And then why don't we also go ahead and update the timeout to match. So we'll use Minutes of 120. So there are two things left to note here before we wrap this up and try it out in Zeppelin. The first is that we no longer need this print statement at the end because we already have an action, which is the foreachRDD on the ActivityByProduct aggregation. And then the second thing worth nothing here is a simple optimization. So notice now how we actually use the activityStream to perform a statefulActivityByProduct aggregation and also use the same activityStream down below to create the statefulVisitorsByProduct aggregation. So this is a DStream very similar to an RDD that ends up having a split of two operations that we're performing against it. So that means activityStream is a very good candidate for being cached. So let's go all the way up to where activityStream is defined and then simply add a. cache. So, remember, when you have a fork in your processing where each branch ends up in an action, then that's a good candidate for you to consider to cache your processing. So let's switch to Zeppelin and see this in action.

Evaluating Approximation Performance with Zeppelin: Demo

In this clip, we're going to demonstrate using stateful transformations in Zeppelin and specifically using the HyperLogLog example. But, first, let's take note of what is different between the Zeppelin notebook and the code we just wrote. So the first thing you'll have to know is being able to load the Algebird dependencies. And in older versions of Zeppelin and certainly versions that we've actually used throughout the course, we used to be able to load dependencies using the %dep keyword. However, this has changed, and if you notice, I actually have an error here specifically because I did upgrade the version of Zeppelin that we're using, and I will actually go back and note on the other notebooks that we've created how to fix this. But for now, the way to fix this is actually through the interpreter. So you go to the Interpreter menu where essentially all the interpreters are available. And if you scroll down all the way to the bottom, you'll see the Spark interpreter. And within the Spark interpreter, you can go ahead and edit the interpreter, and then Zeppelin gives you an option to add the dependencies that you need. And so we go ahead and add the Twitter, Algebird core library with Scala 2. 11. So adding a library dependency this way essentially makes that available to every notebook that you use. And there's good reason for why Zeppelin did it this way, and that's because before it creates the Spark context, it actually always needed you to load the dependencies upfront, and that always caused some headaches for you actually having to go back to the Interpreter's menu and having to restart the Spark interpreter. So instead of doing it this way, you just go ahead and define it upfront and let Zeppelin know what you intend to load. So we actually no longer need this step, and you'll notice that I actually have it disabled. But I'm keeping it here for reference. Now if I scroll quickly, you'll see that the case classes are defined so nothing special here. Again, we're also importing Spark Streaming, and we have a transient Spark StreamingContext variable defined. And, again, the rest of the code is very similar to what we've seen earlier, so I'm going to skip through loading the data from the file and go straight to where we have our functions defined. So one thing you'll notice here that is different between the code that we've written and the code here is that the functions object is now a case object. And the reason I did that is simply to make this functions object actually serialize. And that's, again, something that has to do with the way Zeppelin's actually going to load your code. Most of your code needs to be able to serialize, and adding case to a class or an object actually makes it serializable. So it's a quick shortcut for you to get serialization for your class or object in Scala. Moving on to the next step. So in this step, we essentially do the same thing except that I'm actually removing the reduceByKeyAndWindow because we don't want to wait for 30 seconds to see some results. We are interested in seeing the results immediately, well, almost immediately because our batch size is 4 seconds. And then because I am going to be running the query quite frequently, notice here that after I register the TempTable, I'm actually calling the sqlContext. cacheTable so that the table remains cached and I can constantly recreate it until it gets refreshed without taking a hit in performance. So in addition to what we're doing with HyperLogLog, I went ahead and thought that it would be useful to see a comparison between what HyperLogLog would give you in terms of unique counts and what the naïve approach, meaning you simply keep everything in memory for the entire period and run through calculations that way. So this bit of code here essentially takes the same activityStream, maps over, and produces the same product and timestamp\_hour and visitor, and then runs a simple updateStateByKey operation. However, notice that the state that we're keeping is a sequence of strings. So we're essentially keeping a unique set of every single visitor in a sequence for each key. And we also carry along the timestamp so that we can perform the timestamp calculations that we need. So the end result of this updateStateByKey operation is that we have a key, which is the product and timestamp, and a unique list of visitors, which we maintain completely in memory. And we constantly add to that visitor list with each state iteration. And then what we return is the unique count. And because this is a sequence, a sequence that essentially only keeps unique keys, we can simply return the sequence length. And that gives us a unique count. So we're actually returning the precise unique by virtue of actually saving every single unique value as opposed to an approximation of the uniques. And so, finally, we also register another TempTable called VisitorsByProduct. And so in our query here, I took an example of a couple of products. And, hopefully, we'll see those in the result. We have a query that essentially goes against the HyperLogLog table. Again, a simple timestamp conversion produces the product and the unique\_visitors as unique\_visitors\_hll, and then 0 as unique\_visitors so we have an opportunity to produce the actual unique\_visitors from the union query. And so the second query is essentially a union all with a select that's essentially the same thing except that we reverse the equation here, so 0 for unique\_visitors\_hll and the actual precise count for unique\_visitors from the VisitorsByProduct table for the same set of products. So I'm going to go ahead and start the LogProducer in the background, and I trust you know how to do that by now. We went through a couple of examples of how you simply run that Scala class. So I'm actually going to run that to start producing some records for us. And I already went ahead and also ran all of the previous segments of this worksheet so we should be able to now run this query and start seeing some results. So I actually have the LogProducer currently running in the background. And there we go. We start getting some results, and you'll see that there're two different shades for each color. The darker shade is the HyperLogLog value, and the lighter shade is the actual precise counts, and you can see that the HyperLogLog so far is doing a good job of getting an estimate of one unique record for this time period and these products. So we'll go ahead and run the query one more time. And you'll see now that our snapshot is actually building, so we're not actually losing any data yet because we're keeping a snapshot for an entire hour. And so far, the HyperLogLog estimate has actually been still pretty good. It's still calculating the correct number for each one of these, for example Gatorade 3 uniques at 5:51, and the actual precise count is also 3 uniques. So let's keep running this and see if we can actually get a deviation between HyperLogLog and the precise count so we can demonstrate that. Still pretty accurate on every count. So now you see there actually is a deviation here. So the HyperLogLog in this scenario actually estimated that the Trident Gum had only 3 unique visitors where in reality it actually had 4 unique visitors. HyperLogLog isn't very efficient when it comes to very small numbers, but the tradeoffs are fantastic. You get very small memory footprint, and it's still a very, very accurate representation of your values.

Summary

In this module, our primary focus was on enhancing our experience with Spark Streaming by expanding on what we can do with a streaming application. We looked at using window operations alongside stateful transformations. We saw how updateStateByKey and mapWithState offer different semantics for solving a similar problem and how mapWithState achieves better performance by not having to traverse the entire state for every batch. We also looked at using Twitter's Algebird library in our streaming application to use HyperLogLog for calculating cardinality, and we compared that to the results of an exact calculation albeit on a small set of data. The Algebird library has more sketch algorithms that are widely used in production in many different companies. So I encourage you to take a look at the ones available. For example, CountMinSketch can be used to count the frequency of the different types of events in a stream. So this concludes this module. In the next module, we'll start working with Kafka and producing real-time streams to the Kafka instance provided in our VM and actually start gluing everything together from the batch and streaming layers. See you there!

Streaming Ingest with Kafka and Spark Streaming

Introduction to Kafka

Hi! My name is Ahmad Alkilani, and welcome to this module on Streaming Ingest with Apache Kafka and Spark Streaming. This is probably my favorite segment because of what Kafka enables and brings to the table. In this module, we're going to use the Kafka instance running on our VM, and we'll modify our LogProducer to create a Kafka producer that now truly sends a real-time stream of data to Kafka. A lot of what we've been doing so far has been in setting up for this module and the next one on Cassandra. We'll look at the different ways Spark Streaming integrates with Kafka, and we'll start integrating our batch and streaming code. So we'll really start to see how all that comes together. But let's first introduce Kafka, what it is, where it fits, and some of the architecture behind it because it will influence what we do in Spark Streaming. We'll start with a definition of what Kafka is and then move on to another definition that comes from LinkedIn, the creators of Kafka, that clearly explains what it's meant to be and its place within a company's architecture. So Kafka is a distributed publish-subscribe messaging system. It can be thought of as a distributed commit long where publishers publish data to it and consumers read data off of it. There are attributes, however, that differentiate Kafka from a normal queue, and we'll get into that in a second. Every Kafka server instance is called a broker, and it's a distributed system so you'll tend to have more than one, so three, five, etc. So odd number of Kafka brokers. An odd number is used because it gives you the best fault-tolerance while avoiding a tie in decision making. So in the three-broker scenario, you could lose one broker, and while still having two brokers up, a vote of confidence from two out of three is good enough to keep the system healthy because that represents a majority from the original count of three. Compare that to having four brokers, then you'd need three to gain a majority vote, so you still end up only being able to handle one broker failure. And you can extend that analogy as you go up in numbers. Kafka brokers are meant to scale availability and performance very similar to most distributed system characteristics. Adding more brokers to a cluster can give you higher availability, obviously depending on how much you plan to replicate data across brokers, but also gives you scale. In many cases you'll see either three or five brokers in production deployments. But let's really get into what Kafka is meant to be and how it's different from other systems that could be seen as playing a similar role. LinkedIn defines Kafka as a unified platform for handling all the real-time data feeds a large company might have. So that inherently means that Kafka can handle many different sources of data be it an application emitting events, servers sending logs, or systems sending telemetry data. Kafka's built to handle heavy load. One of the things that distinguishes Kafka from any other frameworks is that it's fast. It's very fast. But even more important is that it can handle multiple consumers reading off of the same topic without a hitch. That's one very distinctive feature of Kafka that separates it from the crowd being able to serve the same messages to a different clients, each client consuming on their own pace. So how's this different? Well, Kafka doesn't act as a queue. That means that consuming data off of a Kafka topic has no side effects to the data itself. In other words, reading a value doesn't remove it from the queue. So essentially Kafka doesn't dequeue values in that fashion. Rather, it's left up to the consumer to decide how it wants to track its own offsets, which brings me to another point. Each consumer tracks its own offsets. It's not the job of the Kafka broker. Kafka provides facilities to ease that operation. However, it's still something the consumer deals with, so that's one thing the broker doesn't have to deal with itself. And so by that definition, Kafka doesn't push data to consumers either. It's a pull model. And another important distinction specifically because Kafka doesn't act as a queue and can serve multiple consumers for the same data is that the operation isn't taxing. It's actually expected that a Kafka topic will have multiple interested parties. And Kafka can deal with that without a problem and no impact on performance. While Kafka doesn't behave as a queue, you can still simulate that behavior with a tweak of some settings for a consumer if you wish to do so. So any given consumer may choose to have queue-like semantics for its entire application, meaning consuming a message would appear to other consumers of that same application as if it were gone and removed off of the queue. But this is achieved through the control of groups and offsets, and we'll get into this in more detail later. So as you can see, there's a vast list of consumers you can plan for data going into Kafka, and that's what makes LinkedIn see it as the backbone for handling all real-time data needs. So in short, Kafka is a publish-subscribe distributed commit log that separates segments of data through topics and allows for multiple consumers of the same data, while maintaining performance levels throughout the cluster. Let's look at Kafka's architecture in a bit more depth so we can better understand how data is produced and consumed. Then we'll start looking at building our own Kafka producer.

Kafka Broker

Let's look at the structure and semantics of how a Kafka broker works and how data is produced to Kafka. For simplicity, we're going to use two Kafka brokers here and examine what each broker might look like as we walk through this example. To use Kafka, the highest level abstraction is called a topic. You can think of a topic in Kafka as a table in a database or some logical abstraction of the data you expect to send to this topic. Although nothing prevents you from sending whatever you want, and you'll have to employ some company standards or safeguards against that, like using some sort of schema registry, the general idea is that you would define a topic and agree on the data that you expect to see in that topic. Every topic has a name, so we'll call our topic WebLogs. Along with every topic, you also define the number of partitions for this topic. Let's choose two partitions for our topic. Every partition in Kafka is called a replica because Kafka supports data replication. Notice, however, how we're distinguishing a special name for these first two replicas, and we're calling them leader partitions or primary partitions. And I will use those two terms synonymously when discussing Kafka partitions. All replicas are treated equal in the sense that they can perform similar duties. However, only one broker can be designated as a leader for a partition, and that has important semantics in terms of reading and writing data to Kafka. All writes start with leader partitions. And all reads only read from leader partitions as we'll see shortly. Also, notice here how leaders are typically chosen and distributed among all of the brokers. So Kafka will try to spread leadership of partitions across multiple brokers for performance and risk aversion. And we'll look at how this affects performance when we look at consuming from Kafka. So we called every partition a replica. And as that name implies, we have replica partitions, and we get to choose the replication factor for the topic. This is very similar to a replication factor setting in Hadoop and allows you to choose how many replicas in total to create for your previous messages for fault-tolerance. Note that this only affects fault-tolerance and does not have any impact on consumer performance. It does, however, have a potential impact on producer performance. And, again, we'll see how this works in detail shortly. So let's choose two replicas just for this example, but a very common value for this would be 3, similar to HDFS. So we'll bring in our replica partitions giving us two replicas in total for each partition. Notice how the replica backups for each leader are on a different broker. This is part of how fault-tolerance is actually achieved. Now the number of partitions you choose has some very important implications. We won't get into all the details here because we simply can't cover everything Kafka has to offer, but it's important to understand that there are implications to what you do with some of these settings. For the number of partitions, this has a direct impact on the maximum allowable parallelism you can achieve when reading from Kafka. So if you only have two partitions for your topic, then only two consumer threads can read from that partition. I'm using the term 'thread' here very loosely because consumers can use multi-threading or create a group of individual processes running perhaps on different systems with one or more threads each that all combine to create a single consumer application. So when we talk about maximum allowable parallelism, we mean per consumer application. Remember Kafka excels at allowing multiple consumer applications to consume messages from the same topic, so that still works. However, the number of partitions defines the parallelism by which each one of those independent applications can consume messages from a topic. So you can have a spread of different types of applications. And within each application, the maximum allowed parallelism is defined by the number of partitions that you've defined for your topic. We'll focus more on consumers in the next segment, so let's continue to focus on the broker. There are more implications to the number of partitions chosen, and we'll discuss one here which relates to the broker and how Kafka works. Remember, Kafka prides itself on being able to handle multiple consumer applications each with its own record of where it's reading off of the topic. So that means that Kafka doesn't dequeuer message off of the topic partition as in when they're read. Rather Kafka in its simplest form of operation will simply retire old messages based on its configuration or possibly an override per topic. So data doesn't live indefinitely in Kafka. It's typically meant to act as a streaming buffer between producers and consumers, in addition to allowing applications to send data to one place and consume data from one place instead of having a spaghetti of connections between systems. In any case, the end result is that data typically eventually rolls off of the partition. The default setting is actually set to seven days. And I say typically because Kafka can keep data indefinitely in an operating mode called compaction. But I only mention that for completeness. If you're considering Kafka for your use cases, just know that it can support a use case like that as well. We're just not going to go into the details of that in this course. So the decision you have to make about the number of partitions is important from an administrative perspective because an entire dataset for a single topic has to be able to fit on a single broker. It's simply how it's physically stored. So you can split your data across brokers by selecting multiple partitions, and each partition has to fit within a single broker for however long you choose to retain that data. If it doesn't fit, then you can either get bigger disks, reduce the retention period, or change your partitioning strategy. When Kafka creates partitions for your topics, one of the partitions of all of the replicas is designated as a leader. The leader replica is where data is written to first and is ultimately read from. Backup replicas server exactly as what their name entails, as backups in case the broker of the leader replica goes down. You have a backup to read from, and one of the backups is elected as primary. Given Kafka is a distributed system and it needs to store and negotiate metadata, for example topic information in which replica is a leader for a topic partition, it relies on a system called ZooKeeper to help manage the negotiation between brokers for making these decisions and storing information. So the brokers communicate with ZooKeeper, and that's how they also learn of each other's existence. ZooKeeper is yet another distributed system. It typically has a three-node setup. And it's used to enable this type of configuration management. Let's reset this illustration and look at Kafka from a producer's perspective.

Kafka Producer

When producers connect to the Kafka brokers, they ask for information about the topic they plan to send data to, specifically the number of partitions and the list of which broker holds the leader replica partition as data's only produced to the leader replica, so the producer needs to know that information. The producer then employs a partitioning scheme to determine which message goes to which partition. Every message in Kafka has a key and a payload value. As the creator of a producer, you can choose the partitioning strategy you wish to employ. So ultimately each message sent to Kafka already knows its destination and targets that broker and leader replica. As far as partitioning strategies go, you may choose to partition the data based on a key. Or if no key is provided, then it's likely you'll choose a random partitioner that elects a random primary partition for the data you produce and changes it periodically. Again, we're not covering all the details of Kafka in this course. However, do note that the random partitioner is what I see many choose if you don't have a good key with a good distribution of values. And while it's a valid choice, it does come with some performance considerations, specifically if you only have one producer sending data, then data ends up filling up a single partition at a time, which means your true parallelism on the consumer end is affected by this as well. It's not so much a problem if you have a lot of producers sending data and selecting partitions at random. So our producer here has a message to send. It has the information it needs from the brokers on the topic and where the leader replicas are for each partition, and it uses a partitioner to determine where this message needs to go. Let's say it chooses P1 as the partition, so it sends it to P1. Now the broker carrying the second replica of P1 then gets a copy of the new message. In a similar fashion, if the producer determines a new message is to go to P2, then the broker with P2's leader replica gets the message, and it's replicated over to the other brokers based on the replication factor specified. Simple enough. Now the producer itself has some control over what it deems as a successful send operation, and when it actually receives acknowledgement from the broker for that operation. This setting essentially allows the producer to fine-tune what kind of guarantees they require, and it's called request. required. acks. A producer may choose to have all replicas sync the data before a successful acknowledgement is received. It can choose to receive acknowledgement after only the leader replica receives the data so a bit less stringent about requirements in favor or performance. And it can also choose to fire and forget and not wait for any acknowledgements at all. Again, this is up to the producer to choose a strategy. As of Kafka. 9, producers are all implemented in an asynchronous fashion. However, you can block and wait for an acknowledgement if you really need to. But in most cases, you can simply opt to act on the callback you get back as a result of receiving that acknowledgement asynchronously. Also as of Kafka. 9, producers by default will send messages as soon as they get them as long as the underlying pipe can handle the traffic and will automatically batch messages if it can't send single payloads fast enough. So that's Kafka's internal structure and a brief about producing data. In the next segment, we'll discuss Kafka consumers, and then we'll work through our LogProducer to send data to Kafka.

Partition Assignment and Consumers

Let's build on what we've learned about partitions, data placement, and replication to better understand how Kafka consumers work. In this example, we have two topics defined called weblogs and telemetry. If I were to tell you that the superset LR annotates leader replicas, then you can probably tell that the weblog's topic in blue has three partitions and no backups, so the partitioning count is three and replication factor is one. The telemetry topic, on the other hand, also has three partitions, but the replication factor is two because each partition has a primary leader replica and a backup. Kafka defines consumer applications in groups called consumer groups. This is a property of your consumer when you create it. The concept of consumer groups allows for different approaches when consuming data from Kafka topics. The most common is that you would create multiple consumers, for example by spinning up multiple instances of your application, where the consumers all share the same group name. And as a result, they share the load from multiple partitions. These consumers can be on the same host, different threads in the same process, or different processes on different hosts altogether, which is a testament to how flexible Kafka actually is. It's very typical to have multiple consumers within a group each on a different host that collaborate to consume data from topics. Let's see how our little example here works. Let's say we're consuming from the telemetry topic. Remember, only leader replicas are used to read data from. When a consumer group comes online and starts consuming data, a partition assignment process happens. Partitions are assigned to consumers within a consumer group. So there's this concept of ownership between consumer members within a consumer group and each partition. So there's a 1 to 1 mapping between a leader replica partition and a consumer within a group. Once this assignment process happens, consumers can start to pull data from the respective partition. So notice how the consumers within a group collectively collaborate to read data off of topic partitions. And, again, a partition's only ever assigned to a single consumer within a consumer group. That means if partition P1 was assigned to consumer C2, then consumer C1 will never receive data from partition P1 as long as consumer C2 is still alive and well and holds ownership of that partition. So keep in mind there's ownership involved. Only one consumer can own a partition. This will help us understand how to achieve semantics that are slightly different than a load-sharing scenario. So what happens if consumer C2 dies? Well then C2 no longer owns partitions P1 and P2, and a consumer rebalance happens reassigning the partitions to the remaining consumers in the group. In this case, we only have C1 left, so C1 assumes ownership of all three partitions and life goes on. So Kafka not only enables fault-tolerance on the brokers but also allows you to build applications that can scale and have fault-tolerance semantics if a process dies. Everything is taken care of for you. So let's take another example. Let's say we have another application that needs to consume data from the same topic. This application is deployed on four hosts, and they all share the same consumer group name called Consumer Group B. The same assignment and ownership process happens as before, and consumers from the consumer group are assigned to topic partitions. Notice, however, this time we have a consumer that isn't doing anything. In reiterating that partitions are only ever owned by a single consumer, so having four consumers in a group in this case doesn't increase the level of consumer parallelism, which should really drive the point about the number of partitions and maximum consumer parallelism we briefly discussed in the previous clip. Because we only have three partitions, a maximum of three consumers within a group can read off of those topics. You can obviously have a group of more consumers. However, they will stand idle until one of the other active consumers dies, then they will gain ownership of the affected partitions. Finally, notice that we're talking about maximum parallelism within a single application or, rather, within a single consumer group. You can define multiple consumer groups that read off of the same topic as we do here. The distinction between groups is that each group is tracking its own set of offsets from where it's reading off of the commit log. So although these groups read from the same topic, they don't collaborate or know about each other in any way. And each group gets a copy of the messages in the topic on its own pace. Let's take a closer look at how that's dealt with. So this example assumes a topic with three partitions. And we're just assuming a single broker here for simplicity. Producers continue to send data to Kafka, and Kafka commits the data to its log. As consumers come online within consumer groups, each consumer group and as a result each consumer starts tracking its own offsets. Where offsets are tracked is really an implementation detail. Kafka doesn't care where you do that. It's up to you. However, the consumer API does provide convenience methods that allow the API and consumer to automatically commit offsets for you on a cadence. In earlier versions of Kafka, the default convenience implementation for this interval-based offset commit was to store offsets in ZooKeeper. This had its own set of problems because ZooKeeper was then required to be able to handle a little bit more load than it's typically accustomed to. So organizations with large implementations ended up having to set up ZooKeeper servers with SSD drives to keep up. In more recent Kafka versions, you have the option to choose where the auto-offset convenience method stores its data. You can still rely on ZooKeeper, or you can choose to store offsets in Kafka itself. I won't go into the semantics of how that works internally, but it basically lets you offload offset commits to a special Kafka topic with compaction-enabled, and Kafka obviously is built to handle millions of messages, so throwing a few offset commits at it periodically is no big deal. Notice that these two implementations are available for use when enabling auto-offset commits. However, you can choose to commit offsets whenever you want and to whichever store works for your scenario. The rule of thumb here, however, is that if you need absolute exactly-once semantics for your application, then you're going to have to manage offsets on your own because only your application can tell if it successfully processed that message and handed it off to whatever downstream system it needs to go to. If you don't need that kind of requirement, then using one of the default auto-commit implementations is the way to go and preferably using Kafka as the datastore instead of ZooKeeper.

Messaging Models

Now that we know how Kafka treats consumers and the concept behind consumer groups, let's have a look at how easy it is to achieve different messaging semantics or messaging models. Specifically, we'll look at the case for a publish-subscribe model and a queueing model. In a publish-subscribe model, all consumers of a topic are expected to receive the exact same set of messages, meaning every consumer gets a full replica of all messages sent to the topic. To achieve this using Kafka's consumer groups, you'd essentially have one consumer per consumer group. So each consumer group ends up with a single consumer. So, for example, we could have consumer C1 as part of Consumer Group A, and because it's the only consumer within that group, it receives messages from all of the partitions. There's no one else to share those messages with within that consumer group. Similarly, if we wanted to spin up an additional subscriber, C2, then we give it its own unique consumer group called Consumer Group B, and C2 is the only consumer within that group. So, again, it acts as a subscriber to the topic and receives all the messages the topic is sent throughout all of its partitions. What we end up here with is that C1 and C2 are both subscribers to the topic, weblogs, and receive a copy of all of the messages weblogs is sent. And you can keep adding additional consumers with unique consumer groups for every new subscriber you need. Let's take a look at the queueing model. So a queue model, on the other hand, is achieved using the multiple consumers per consumer group approach. Remember, a consumer group is just a property value that the consumer specifies. So if C1, C2, and C3 all indicate their group name is called MyQueue on startup, then they'll all share a portion of the data. The fact that there is ownership of partitions, meaning every partition is only ever assigned to a single consumer within a group, this means Kafka guarantees that data that goes to partition P1 is only seen by C1, and similarly for P3 and P2. In other words, using consumer groups this way means that a message sent to a topic is only ever seen by a single consumer within a group. So this will essentially have Kafka appear as if it were a true queue. And this approach is very typical for Kafka applications. However you have the flexibility to mix and match whatever you want. All it takes is setting the consumer group property for the consumer. In the next segment, we'll start writing some code and create a Kafka producer by modifying the LogProducer we've been working on so it submits messages in a stream directly to Kafka.

Kafka Producer: Demo

So we have our project back up, and we're going to work on changing our LogProducer to produce to Kafka directly, so a stream of messages straight to Kafka instead of simulating a stream by sending data to a flat file. So the first stream we're going to do is look at our project's POM file and make sure we have the proper Kafka dependencies. So if we search for Kafka, we'll find that we're using the org. apache. kafka/group and the kafka-clients/artifact with the. 8. 2. 1 client version and also the dependencies for Kafka and Spark. So you'll notice we'll have the spark-streaming-Kafka and then 2. 11 indicating that that's the Scala version that we're using. So now that that's all taken care of, I'm bringing up the LogProducer. And I've actually already made changes to the LogProducer, and I'll walk you through what changed to actually send data to Kafka straight. So the first thing you'll notice is that we have a set of imports. So we're importing KafkaProducer, Producer, its config, and an object or a class called ProducerRecord. And then the beginning of the code doesn't change up until we get to a point where we need to define some properties that relate to our Kafka broker instances, in this case only a single instance that's running on localhost, essentially the VM that we have up and running. And also which topic we want to send data to and the specific topic configurations. So we'll start off by creating a topic variable called topic with the value weblogs-text. And you'll notice that I actually have these values hardcoded just for clarity. However, it is best practice to use the application configuration file as we've done in an earlier module. I will actually move this to the Properties file just to showcase how you can do that again. But we'll leave these other properties here for clarity. So to produce messages to Kafka, we need an instance of a KafkaProducer. And a KafkaProducer takes a set of properties so that it knows which Kafka cluster to talk to, essentially the broker's addresses, so we have a property called BOOTSTRAP\_SERVERS-CONFIG. And, again, this is localhost:9092. Now this is a Bootstrap Server configuration or in other systems known as a seed config. And ideally you'd have a list of brokers for your value. But you don't actually need the entire set of brokers. So you could do with one, but it's better to have more than one in a comma-separated list. But the general idea is that once you're able to reach one of them, it will provide you with information about the entire cluster. So all you need to have is enough information to reach one of the brokers. And we say it's better to have more than one configured here because if you only had one configured, and that one just happened to be down, then your program won't be able to start up. This is usually done through some sort of service discovery. And, again, in our scenario, our Kafka broker is already up, and we only have one instance of it running on localhost port 9092. Now another thing you'll notice is that props is just an instance of a Properties bag, so essentially a map of key value pairs. And while you will find documentation about what the actual key values to use for specific configurations, for example, the actual configuration key to use for BOOTSTRAP\_SERVERS or the other configurations that we're going to use, however, these keys actually changed between different Kafka versions. So you're actually better off going against the ProducerConfig static class and just picking up the values of the static defined configuration variables so that you know your code won't have any problems in the future. So for the BOOTSTRAP\_SERVERS\_CONFIG, this is the one that we're picking up. And, similarly, we are going to pick up a key for the KEY\_SERIALIZER and the VALUE\_SERIALIZER. And in this case, we're actually just going to be sending data in a string format, so we'll provide the classes org. apache, Kafka. common. serialization. StringSerializer for both. And then we provide the ACKS\_CONFIG. Remember, it's up to the producer to choose what a successful message send means. So this is where you want to configure your producer behavior, whether you want to do a fire and forget, or if you're simply okay with the first broker, essentially the leader, accepting your message and acking back telling you that it received it, or that you want to make sure that all replicas that you have defined on that topic have received your message before you receive an acknowledgement. And then, finally, it's good practice to provide a CLIENT\_ID\_CONFIG. This really doesn't change anything in your program except that if you have any exceptions or you're looking at logs, this CLIENT\_ID\_CONFIG will carry along so it will help you figure out from which client your data is coming from. So we used this set of properties to create a KafkaProducer. And notice that the producer takes two types, the type of the key and the type of the value. In our scenario, we're actually not going to provide any keys, so we're providing nothing. This is very similar to null in Java. And then the properties that we've just defined. And we end up with instance of a Producer with the same type. Now remember how the producer actually communicates with the brokers. So you can actually start asking it for specific information about your topic. So you can use the KafkaProducer to give you information about the partitions for a specific topic. So that's what we're doing in this example just to illustrate this concept. So now we enter a loop to produce data to our Kafka topic. And because Kafka is a very high-performing piece of software, we're going to cheat a little bit and continue to simulate the latency that we're introducing between messages. So we're still going to use the number of files, variables, and the records per file even though we're not producing to a file. We're just using that so that we can introduce a sleep timer between iterations. So you'll notice the FileWriter is now commented out. We don't use that anymore. And the rest of the code here is the same. We're just producing random messages. however, when we get to a point where we have our line, so this is what we actually want to send, this is the string that we want to send to Kafka, then we create a new instance of a ProducerRecord. Each ProducerRecord takes the topic that you want to produce to. So that means that you can actually produce to multiple topics using the same KafkaProducer instance. As long as the serializers are the same and the Kafka brokers are the same and all the other settings that you care about regarding that topic are the same, you can actually produce to multiple topics if you want to if the topic is associated to the ProducerRecord itself. And ProducerRecord actually takes multiple overrides. So if I Ctrl+P, you'll see that it can take a specific message where you're targeting a specific partition that you know of, so you can decide to spread your data to partitions based on some intrinsic knowledge that you have or rely on the default partitioner, which essentially performs a hash on the key value. Now there's something really important here that you have to know about key values. You want to make sure that you're using the function that corresponds to what you're sending. And what I mean by that is if you don't have a key, which in this case we don't, then don't make a call to the function that provides the key essentially either the first or the second function calls, and provide a key value of null. What you're telling the KafkaProducer to do in that scenario is to send all of your data to the hash of the null value so everything ends up going to a single partition. And you're not getting any parallelism, and that broker and partition are getting the results of all of your data produced from your application. So what happens in the scenario where you're not providing a key at all? You either provide the partition that you want to produce to manually, or you provide no key at all, and you let the KafkaProducer select a random partition that it will produce to for a certain amount of time, usually that's about 5 minutes, and you can control that as well in the properties that you've defined. And then when that timer hits again, it will go and randomly pick another partition. So it is sending to a single partition at a time instead of multiple partitions, but it will constantly randomly flip between different partitions. So now that we have a producerRecord, we can go ahead and use the KafkaProducer and simply send our producerRecord. Now if you wish to act on any errors that you get back, the send function actually provides you with means to provide a callback where you can take action on failures and try to resend, although if all you're doing is resending a failed message, then you may want to just increase the resend account on the properties for the KafkaProducer, and the client will take care of that for you. However, if you have different actions to do, for example, you need to notify another system or you care to actually log locally that a failure occurred, then you can implement this callback and perform those actions. However, this isn't a full-fledged course on Kafka, so I'm going to leave that for you to exercise. And then we simply iterate and then provide a random sleep. And then towards the end, we want to make sure that we close the KafkaProducer. And, again, ideally you want to have a try-catch here and perform this close in a finally block. That is the way you should be coding. Also, close performs two actions in Kafka. 8. It actually closes the KafkaProducer, but it also blocks until whatever messages the KafkaProducer has in its buffer are actually sent out. And in later Kafka versions. 9 and. 10, they separated those two functions so you'll have a KafkaProducer. flush, which as its name implies basically waits to flush all the data out, and then you also have a close method to close the connection to the Kafka brokers. So that's really all it takes to produce data to Kafka. It's not that complicated. In the next segment, we're actually going to run this code and bring up Zeppelin, write a little bit of code in Spark to consume the messages that we're producing. But before we do that, let's go ahead and change the hardcoded topic name. I've actually already gone ahead and added the required configurations in our configuration object. And so now we have a KafkaTopic variable. So let's go ahead and take a look at that. So the KafkaTopic variable just brings up the weblogGen, and it asks for a string called kafka\_topic. And if I bring up the application. conf, you'll notice that the kafka\_topic configuration is there, and it's called weblogs-text.

Spark Streaming Kafka Receiver: Demo

In this demo, we're going to use Zeppelin to illustrate Spark Streaming using Kafka as a the source. And we'll illustrate that using bits of the code that we've written in previous modules, in addition to the producer that we've written in the previous clip. Before we get started, we need to make sure that Zeppelin has the proper dependencies for us to be able to use the spark. streaming. kafka utilities. And so to do that, we have to add the appropriate dependencies by going to the Interpreter section, which I have opened already here in a separate window. And if you scroll down all the way down, you'll find the Spark section. And you'll want to edit that and then make sure that you add the Kafka dependency and the spark-streaming-kafka-assembly dependency. Spark-streaming-kafka actually comes in two flavors. One is the spark-streaming-kafka-assembly. And one is simply a plain spark-streaming-kafka. The assembly dependency makes sure that you have everything that you need including all the other additional libraries, and it doesn't assume anything about your environment. So it essentially gives you an uber jar to deal with, and that's really what Zeppelin needs for this to work. So we'll switch back here, and you'll notice that we have our import set up. One very important note about the imports. There is a conflict between the namings of the spark. streaming. kafka package and the actual Kafka package straight up. And the way to deal with that is to basically start the Kafka import with \_root\_. And that basically prevents any Class Not Found exceptions that you might see because the import couldn't be handled. So apart from that, nothing special here. We do start with a set of properties that we need for our consumer. And in the consumer's case, it actually needs ZooKeeper's connection string and not Kafka's connection string. So zookeeper. connect, and we only have one instance running in our VM. So only one instance is provided. And it's running on port 2181. Now there's also the all-important group. id. And this is really what marks this instance of a consumer as part of a group. So if I were to launch another Zeppelin notebook and use the same group Id, then both of those processes will actually coordinate to consume from the topic. Then we have an auto. offset. reset setting, which applies to a normal consumer that is relying on ZooKeeper to receive its data, and that indeed is the case here because we are going to be using the Spark receiver-based Kafka consumer. And then we'll take a look later at the direct-based approach and explain the difference between the two. So auto. offset. reset, largest, means that we are only interested in receiving the latest data as and when we launch our application. And so we have the case class Activity defined, and all we're going to do, and you can see that I've already run this first segment, all we're going to do is create a Kafka stream, which I'll walk through in a second. And then apply the same transform that we've seen many times before. However, towards the end of it, we will create a DataFrame and register a table called Activity and also cache that table using sqlContext. cacheTable so that we have an opportunity to query it. So how do we use Spark Streaming to consume from Kafka? Well, Spark Streaming already comes with a set of libraries that help you consume from different sources, one of them that's readily available is Kafka. So all we need to use is KafkaUtils, which comes as part of Spark Streaming. And then we create a stream for defining the types of the key and the value. And then we're saying that these are the decoders that are responsible for decoding these values, again, it makes sense that it's a string decoder. And then it also needs the SparkStreamingContext (ssc), the KafkaParams, which we've defined up here, which provide the zookeeper. connect string, the group. id, and the auto. offset. reset. And there are a ton more configurations that you can override, but these are the most important. And then the createStream method takes a Map, so essentially a key value pair of topic and number of threads. So what this is saying is for each topic that you want me to consume from because a consumer can actually read from multiple topics, so you can provide a list of these, how many threads do you want me to use to consume from that topic? In general, it's a good practice to use one thread specifically in the case of Spark Streaming because that thread and that consumer will only land on a single host. And if you spin up multiple threads, you're just contending for resources as opposed to introducing better parallelism because you're still reading from the same NIC or network card, and you're still using the same CPU. So what can you do if you want a higher degree of parallelism? So the approach Spark takes is that you create multiple receivers. So each one of these createStream statements essentially creates a new receiver. So the idea is that you would create multiple streams and then union them all together, and we'll see how to do that later. But for now, we're only consuming from the weblogs-text topic and using one thread for the consumption. And then we specify StorageLevel. This is essentially the receiver where it intends to save data. We give it MEMORY\_AND\_DISK in case the receiver runs out of memory. Now at this point by calling createStream, we receive a receiver input DStream. And so it's just like any other DStream, and we can map against the data. And in this case, we actually want to receive the values and not the keys. We map against the record, and then the record is a tuple of two items, so we retrieve the second item essentially extracting the value from the record that we receive. And that becomes our kstream input, which is now a string line of text exactly as if we were reading it from the file system. So then we can run the transform just as we did earlier and register the TempTables. You'll notice that we have a SparkStreamingContext(SSC). remember, and we set that to 5 minutes. And that's to serve the purpose of this select statement because we have no window function so Spark will assume that it doesn't need to keep around the RDDs for a long time and will dispose of them before we get a chance to run our select query. So you normally wouldn't have to do this because Spark will already know of the select query and perform that operation before discarding any RDDs, so this comes in handy when dealing with interactive scenarios like this. So I'm going to bring up the IntelliJ Id and go ahead and run the LogProducer so it starts producing data to Kafka. And we'll wait for it for a second to see some data produced. So this is sending messages already, and we'll go ahead and run this. And then you should see the print statement actually start producing some records, and we can follow that output. Here we go. So we're actually receiving data from Kafka. And then we can run our select query. And we should see some results. Let's have a look at our StreamingJob and see how it changes to use a Spark Streaming Kafka receiver. So apart from the additional imports you'll need as we've seen earlier in Zeppelin, the changes are pretty straightforward. We have our list of kafkaParams setting the zookeeper. connect string, group. id, and auto. offset. reset as a minimum. And then we simply use Spark Streaming's KafkaUtils. createStream providing the key and value types along with the appropriate decoders, providing the topic name and mapping on the receiver input DStream to extract the value because that's what we care about. And notice that's pretty much the only thing that changes in our application to read data off of Kafka instead of a text DStream.

Spark Kafka Receiver API

In module 2, we discussed Spark's receiver model and how Spark uses a persistent process that consumes a task that is called a receiver. And that process is what's responsible for receiving data from external systems. We also mentioned that there was a direct streaming approach specific to Kafka that doesn't rely on receivers. So let's take a closer look at how the receiver approach differs from the direct approach and also see how we can modify the code we just used to receive data from Kafka and increase the level of parallelism by adding receivers. Spark comes with two flavors for receiving data from Kafka--the receiver-based approach, which uses Kafka's high-level consumer API, and the direct approach, which uses Kafka simple consumer API. So Kafka itself comes with two consumer APIs, the high-level and the simple consumer APIs. The high-level consumer API for Kafka typically uses ZooKeeper to automatically commit offsets for you and doesn't provide access to as many low-level details as the simple consumer API does. So as the implementer of a consumer using the high-level consumer API, you don't have to worry about those details. On the other hand, Kafka's simple consumer API has no high-level abstractions so you're down to the details of how Kafka consumers and groups are implemented. Don't confuse the simple consumer API and assume that it's simpler to use. It's actually not. It just provides simple abstractions, and the rest is up to you. I purposely don't like that name, but that's what it's called. So the high-level consumer API provides high-level abstractions and makes your life easier, and that's indeed what Spark's receiver-based approach uses to consume data from Kafka. As the name implies, sparks receiver-based approach uses receivers to receive data, and the data is stored in Spark executors. The upside to this approach is that it provides lower latency access to the source data because the receivers are always live in consuming data and don't need to be scheduled for every batch. The downside is that it requires a write-ahead log to ensure zero-data loss. And Spark provides that as an option that you can enable where Spark will first write receive data to a log on HDFS. Adding a write-ahead log obviously makes things slower. And in the case of Kafka, it feels a bit redundant because Kafka already is a distributed write-ahead log. So this approach doesn't take advantage of that. The receiver-based approach allows for at-most-once and at-least-once semantics but doesn't allow for exactly-once semantics. The direct approach, on the other hand, doesn't use receivers at all as it queries Kafka on each batch for the offset range. And to do that, it relies on Kafka's simple consumer API. The advantages are a simpler approach to parallelism but at the expense of having to schedule tasks and wait for them to consume the data. So this is definitely at the expense of latency. And by latency, we mean the time an event commits to Kafka to the time your application can act on it. The direct approach, however, has some other advantages in that it doesn't require a write-ahead log for zero-data loss because it relies on Kafka's retention to replay messages. So it's better at processing larger datasets. It also allows for exactly-once semantics, but you don't get that for free. You have to do some work, and your target store needs to play a role, for example, by allowing idempotent upsert-like operations. So let's take a closer look at how we can increase the level of parallelism for a receiver-based approach and then compare that with the direct approach. For the receiver-based approach, the default setup creates a single receiver that is responsible for all the partitions from Kafka. This is precisely what we've done so far in our own code where one of the tasks is designated as a receiver. To increase the level of parallelism, we'll want to add more receivers. So we'll create a list of Kafka streams and union them all together. Let's see how that looks. In this scenario, we increase the consume parallelism by adding more receivers. So we need to create a list or sequence of streams, and we do that in Scala by simply creating a range collection, three items in this case, map over every item, and create a stream. The result of this statement is an index sequence of ReceiverInputDStream. And we can feed that to the SparkStreamingContext union (ssc. union) to get a unified view of all the streams. Now this union is just a meta-operation, so you don't incur any additional cost by doing this. And notice this is a contrived example because if you only had three tasks available and you used three receivers, then you'd have no tasks available to actually process the data. This is one of the things to keep in mind when using the receiver-based approach and also an important distinction between this approach and the direct approach. As we discussed earlier, receivers occupy their task indefinitely. So for any processing to happen, more tasks need to be available. So that's a downside to the receiver approach if latency isn't a concern. A quick Scala tidbit: Notice the underscore placeholder that I'm using here. Instead of what you'd typically see as a variable named declaration for a lambda function like this, Scala lets you substitute the underscore if you really don't care for using the variable. So instead of making up a variable name and not using it, Scala gives you a shortcut to just provide the underscore, which also adds clarity because now you know that the lambda function in here doesn't make use of the variable. So just a quick tidbit on this code so you're not stuck on that point. Let's take a closer look at the direct approach next.

Spark Kafka Direct Streaming API

In the direct approach, notice there are no receivers. Tasks are scheduled and occupied to consume data for the batch, and then the tasks are released for other operations. This is in direct contrast to the receiver-based approach where the receivers always occupy the tasks. So this behaves just like any other task that needs to be scheduled by the driver, and so it's the driver's responsibility to also determine the offsets since the last batch and schedule those tasks for those offsets. The direct approach also relies on Spark checkpointing to save the offsets for each batch. Remember, the data's actually consumed as part of the batch processing itself and not in a dedicated receiver, so it's safe to simply store offsets using Spark's metadata checkpointing on the driver. Each batch, the driver consults Kafka for a list of latest topic and partition offset information for the topics it's interested in and scheduled tasks to start consuming data starting from the checkpoint at offsets and up to the latest just received. For the very first batch, the value for the auto. offset. reset configuration property takes effect similar to the receiver-based approach and the general Kafka consumer concept. From a configuration perspective, you can see that the direct approach doesn't rely on ZooKeeper at all for periodically saving its offset location. And because of that, it doesn't need the ZooKeeper connection string information. Instead, it needs the Kafka brokers to retrieve metadata about the topics and partitions it's consuming from. In addition to that, parallelization is easier and automatically dealt with by the direct stream consumer. It starts off by creating Kafka for the number of partitions per topic and creates a 1 to 1 mapping of topic to Spark partition. And as we know, each Spark partition maps directly to exactly one task. The end result of this is that each Kafka partition is automatically mapped to a Spark partition. And a task is consumed to receive the data. The upside is that you don't have to deal with parallelism and do anything special as we've done by unioning multiple streams in the receiver-based approach. The downside is that you don't get to control the parallelism at ingest, so if your Kafka topic has 20 partitions, but it's really not a high-volume topic, the driver still ends up having to schedule and eventually consume 20 tasks on the executors. So this can certainly be an overhead and, in my opinion, adds a bit too much tight coupling between the Kafka topic setup from an infrastructure and management perspective and how your program executes. Still, the direct approach still comes with benefits for exactly-once streaming semantics without the need for a write-ahead log, and having access to offset information as we'll see shortly is definitely beneficial when recovering from complete application failures or upgrades. The API is fairly similar to the receiver-based approach and is available under Kafka Utils. There's a variation from the version that you see here on the slide where you can provide the function with a map of topic and partition offset information, and this gives you flexibility to control where the application starts from on cold starts. So if you need to upgrade your application code, you can have it start from a known location instead of relying on the auto. offset. reset configuration, as we'll see here in our demos. So let's go ahead and put all this information to practice. In the next clip, we'll look at how our application changes to use the direct approach, and we'll actually start tying in batch processing and saving data onto HDFS so we can feed our batch processing as well.

Direct Streaming API: Demo

Now that we have an understanding of the direct stream approach, let's go ahead and modify our code to use a direct Kafka stream setting ourselves up to use some of the features it unlocks specifically around offset management. And this really shouldn't take long. So the first thing we're going to do is add a new section for the kafkaDirectParams. And note the differences. Again, the receiver-based approach uses ZooKeeper to save offsets, and the Kafka direct stream approach uses the broker metadata and saves the offsets with the checkpoints that the driver makes. So now that we have a set of parameters, we can go ahead and start using them. So we'll create a kafkaDirectStream. And, again, the direct stream functionality is under KafkaUtils. So we'll create a direct stream. And the direct stream also takes the key value types and their decoder types and then Ctrl+P to see the parameter lists. So it needs a SparkStreamingContext, so we'll provide it with that. And notice we're going for the last method override here, so we need to give it Kafka parameters, so the kafkaDirectParams we've defined. And then a set of topics given that we only have one topic. So we'll just give it a set of our topic that we have defined. So let's go ahead and do a little bit of cleanup. So we'll remove the kafkaParams that we don't use anymore. And we'll also remove the creation of the receiver-based Kafka stream and replace that with our kafkaDirectStream. And then everything else again should remain the same except that you'll notice I forgot to map and only provide the values, so let's go ahead and do that. So the value that you get from a Kafka stream is a tuple of the key and value types, and all we're concerned with is the value. So we're now using the direct streaming approach instead of the receiver-based approach. How about we do a little bit more refactoring and clean this up. So we're going to clean up this section. And to do that, we are going to want to have a function that accepts an RDD of string and actually returns an RDD of activity. So we'll take this input, Ctrl+X here, and then go to our functions. And then let's define a function called rddToRDDActivity. And we'll have this function take an input of type RDD of String, and we'll create our function and paste the code we copied earlier. And so now we have a function called rddToRDDActivity, and it takes an input RDD of type String and returns an RDD of type Activity. So now we can go ahead and reuse this. So transform input, and all we're going to do is call functions. rddToRDDActivity and provide our input. So now this should still return the same results. Great! Now you can see how you can actually start abstracting away some of the business logic where the flow of the actions and transformations that you're doing is clear, and it still makes sense from a business perspective what's going on in the code. And this also opens up the door for reuse. So, for example, if another team member needed to use your function, then this could be part of a shared library.

Direct Stream to HDFS

Now that we have a direct stream in place, let's take advantage of some of the features it provides. In this demo, we're going to move on to a step where we start filling in the pieces of the lambda architecture We'll use Spark's direct stream from Kafka to publish data to HDFS to be used in batch processing. And we'll build resiliency into our application. Spark applications in general can recover from executor and task failures and can even pick up where they left off if the driver fails or the application gets restarted by appropriate use of checkpoints. However, Spark cannot recover an application from a shutdown if you need to make a fix and your code changes or even an enhancement that you add to your code. So as of the current version of Spark, code change invalidates metadata. So its recovery from checkpoints, and that includes the direct Kafka stream offsets, is hindered by this process. So if your code changes for any reason, Spark can't handle reading from its checkpoint files, and your defaults come into play, and that's precisely where we need to intervene to make sure things come up where we're not missing any valuable data or reprocessing data we shouldn't be. So what we're aiming for is to store the data in HDFS in such a way that allows us to query it for the purpose of batch processing, but also to solve the resiliency and cold start problem if our code changes and Spark fails to recover from its checkpoints. So we need to store the offsets for each Kafka topic and partition we're consuming from. On HDFS, this might look something like this where we're using the Kafka topic and Kafka partition as high partitioning keys, and the data itself is stored alongside offset information that tells us where this data comes from. Remember, offsets are tracked per partition so it's not enough to track offsets per topic. So we plan on syncing our data to HDFS in that format. But where can we get the topic partition and offset information from? Well it turns out that the direct stream implementation introduces a new class called HasOffsetRanges that extends from the base RDD. And that's actually what you get back. It doesn't show that way at first as not to break any interfaces. However, you can take a direct stream DStream, grab an RDD within it, and cast that to an instance of HasOffsetRanges, and then call the offsetRanges function against it. And this gives us an instance of offsetRanges for the RDD. Now, remember, an RDD consists of partitions, and because each Spark partition maps directly to a Kafka partition when using the direct stream, we can simply use that to our advantage and access offsetRanges by the index of the Spark partition's number. Again, partitions are aligned as long as you haven't performed a shuffle operation and caused any disruption. So let's go ahead and see how we can apply this in our code.

Direct Stream to HDFS: Demo

So we're back in IntelliJ, and we're going to pick up from where we left off the last time where we created a common function called rddToRDDActivity that created the activityStream of type DStream of Activity. And it's a good thing that this operation is cached because we are going to fork the pipeline of operations we're going to do. One pipeline is going to save the data to HDFS and be responsible for that. And the other pipeline is going to continue to do the streaming operations that we've already grown accustomed to. So we want to take this activityStream and store the data on HDFS. So we're going to go ahead and apply a foreachRDD. And for every RDD, we're going to create a DataFrame. So we'll call it activityDF. And we do this so that we can use the Data Sources API against the DataFrame. So now that we have a DataFrame, we can go ahead and use that, and use the Data Sources API so we can use the write method and then partitionBy. And, remember, we wanted to partition by the Kafka topic and Kafka partition. So we first actually need to get that information. So we're actually going to introduce that information to our original activity stream by modifying the rddToRDDActivity function. But first let's examine the actual activity case class that we had. So notice that the case class activity actually had a map called inputProps, and this is when we're actually going to start using. So all we're going to do is add additional properties that we know about this activity essentially telling us which Kafka topic partition from and to offsets the source data came from. And we can get at that information because we're now actually using the direct stream. So let's go ahead and to our functions package. And then the modifications we're going to make are to this function. So the idea is that instead of immediately flatMapping against the input RDD, we are going to cast the RDD and get some information from it. So we'll create a variable called offsetRanges, and we'll set that to the input and cast that asInstanceOf(HasOffsetRanges). Alt+Enter to import that. And then we'll call offsetRanges to get the range information. So now if we examine the type of this offsetRanges variable, it's an array of an offsetRanges. So we know we can do this because we're using the direct stream approach, and the instance of the RDD is actually an instance of the HasOffsetRanges class. And that class has a method called offsetRanges, which returns an array of offsetRanges for each partition. And they're indexed by the partition number. So in order to access the correct range, we need to get that partition index from Spark. And remember that the partition index in Spark is going to have a 1 to 1 mapping to the partition index in Kafka when using the direct stream. So we need a way to get the Spark partition number for each partition within this RDD. And Spark provides a method called mapPartitionsWithIndex. And you can see that this takes a function that expects an integer. And that integer is actually the index number, and an Iterator of String, which is the value type to perform the map operation against. And so mapPartitions is very similar to a map operation except that it runs a partition at a time and provides you with an iterator for each partition, in addition to giving you the index of that partition. So let's go ahead and provide an implementation for this. So we'll call the first field index. Actually, I have to provide a function. And then the iterator i-t. And then we have to provide an implementation for this function. So let's go ahead and get the offset range for this particular partition. So we'll call the offset range or, and then we will get that offset range by going against the offsetRanges for this particular index. And now the offset range itself has information about the fromOffset, the partition, the topic, and the untilOffset, and that's what we're going to use to add to the map of input properties. So now we actually have an iterator for each partition called i-t, and that's an iterator of String. And that's what we're actually going to use to flatMap against. So we'll take all of this code and apply it to the iterator instead of the input RDD because we're already iterating on the records for the input RDD. So what we've done so far is we've taken the input RDD, casted it to an instance of the HasOffsetRanges class so that we can call offsetRanges and get that information. And then we simply take the input RDD, apply mapPartitionsWithIndex. So now we have an index for each Spark partition and an iterator for all the records within that partition, which essentially is an iterator for every line that we have. And then we simply apply the flatMap against the iterator. And during that application, we actually create an activity record. So all that's left for us to do is, again, fill in this inputProps variable, which is currently set to an empty map. So after we bring in everything, let's go ahead and add a new Map. And in this Map, we want to provide information about the topic, the partition, and the offsets. So we'll create a topic key and then give that the value from the offset range. Similarly, we'll create a KafkaPartition key, give that the value from the offset range as well. So that's the partition. Then we'll create another key, call that fromOffset and, again, pull that information from the offset range, and another called untilOffset and pull that information from the offset range as well. And then one last thing, the Activity case class is actually expecting a Map of string value types, as well as string keys, so we'll convert all of these to strings as well, untilOffset, the fromOffset, and the partition number. So now the rddToRDDActivity function actually returns an activity record which has the input property set where we have some keys that represent the topic, the Kafka partition, fromOffset and untilOffset, and their respective values. So let's go back to our StreamingJob. So now we have one more thing we have to change before we can convert the input to an instance of hasOffsetRanges. The conversion to hasOffsetRanges has to be the first thing that happens on your stream. So that means even a simple thing like this. map changes the underlying RDD data type, so we have to apply our conversion and extraction of offsetRanges even before the map we have here. So let's go ahead and remove this map and see what we need to do to fix this. So our input now is actually an RDD of a tuple of String and String, and this is the key and value coming from Kafka. So we have to change our function here to now accept a tuple of two--String and String--and those represent the key and value coming from Kafka. And so we'll also have to do a couple of changes here so the iterator now is an iterator of the tuple of String and String. So for every record we're going to get, this is actually going to be a key value pair. And all we need is to create the line variable and extract the value from the key value pair. So now our function expects an RDD of type tuple of String and String. And it automatically does the extraction of the value for us. So now we can actually continue our code, and this should resolve properly. So now we're at the point where we're going to use this activityStream, foreach against it, and foreachRDD within it actually create a DataFrame and save that to HDFS. But for us to do that, we actually have to extract some of the values that we've created, again part of the activityStream inside of our input properties. So now we have a variable called inputProps. And within it, we have a map of key value pairs that we actually want to extract. And because we have a DataFrame here, this actually becomes a simple operation. So now that we have a DataFrame, we are going to apply a selectExpr, and you can see the inside of our selectExpr. We are grabbing the columns that we want, so timestamp\_hour, referrer, action, prevPage, page, visitor, product. And then we actually extract the value of topic from inputProps. topic and use the column name topic for that. Similarly, inputProps. kafkaPartition and we call that kafkaPartition. And the same thing for the from and untilOffsets. So from this, we learn something new that a DataFrame actually has a function called selectExpr. It's very similar to a select where you're projecting certain columns but with the added flexibility that you get to add a little bit of code inside of each column expression. So now this activityDF has everything we need, all the columns that we want to save to HDFS. And then we're going to simply use the activityDF, write, partitionBy, and we'll partitionBy topic, kafkaPartition, and the timestamp\_hour. And then let's not forget the mode. So we're going to use a SaveMode of Append. So that means every RDD that ends up writing data to HDFS will only append its data. And then we are also going to be explicit that it is of type parquet and provide the HDFS location. So our HDFS location, and we're just calling this weblogs-app1. And I'll select all this and Ctrl+Alt+L just to format it properly. So now we have a stream receiving data from Kafka, and we extract information about the topic, partitions, and offsets. And we're using Spark to also save that data to HDFS. Now this isn't your ideal scenario. You typically would actually want this code to run in a separate program and not as part of your Spark Streaming application. And that's because if you make changes to your Spark Streaming application, you don't want that to interfere with your storage to HDFS. Remember, the lambda architecture splits into two--your batch operations and your streaming operations. And you don't want one interfering with the other. But for simplicity, I just included everything in the same code. Ideally, you'd take this out, split it into its own application. And then even better, you actually wouldn't do this manually yourself. You would use something like Camus or Kafka Connect to actually save your data from Kafka directly to HDFS so you're not relying on code yourself. And those systems like Kafka Connect and Camus already have mechanisms built in to guarantee exactly once delivery to HDFS, so you probably want to utilize those. However, we're keeping it simple here and also trying to teach you something about how to do this within Spark. So now obviously we're taking the activityStream, we're saving the data to HDFS, and then we're still using the same activityStream to create our stateful aggregations and everything that we've done previously. So our streaming application remains the same, and we're just relying on a couple of abstractions now that we have a function to extract the data that we need for us. So let's go ahead and test this. So we're going to package our application. And I'm going to fast forward the video here. Now that the build is finished, let's go ahead and go to the target directory and then copy the shaded jar. That's the one that has all of the dependencies. And this is where you cloned the Git project under the Vagrant directory. So I'm just going to paste here. And then go ahead and bring up Cygwin under the same directory. So I'm using Cygwin now. And if I ls, you'll see that I have these spark-lambda-1. 0-SNAPSHOT-shaded. jar. And then we're actually going to SSH to our VM to run the code. So use Vagrant to do that, so vagrant ssh. This assumes that your VM's already up. So if you don't have it up and running, then you should run vagrant up. So I'm inside my VM, and I am going to sudo su to change to the root user. And then let's navigate to /Pluralsight/spark/, and you'll see that we have an instance of Spark already available for you here. And then we're just going to go ahead and run our application from here. So we'll use spark-submit. And this time we actually want to run Spark locally. So we're going to use local mode because our VM actually is limited on resources, and we don't have enough resources to run this on YARN. If you decide to change your Vagrant file and modify the number of CPUs and the amount of memory available, then you might be able to run this on YARN. But for me, I'm going to run this locally and show you the results. And so the master is local, the deploy-mode is going to be client. And then let's give the driver-memory about 2 GB and the executors about 1 GB of memory. That should be enough for our use case. And then let's just say we have a single executor-core. Again, our VM is pretty scarce on resources, so that's why we're doing this. And then we need to provide the class that we're going to run, and that's under the streaming package, and it's called StreamingJob. And then the path to our Spark jar, and that's under our Vagrant share with the name spark-lambda-1. 0-SNAPSHOT-shaded. jar. So let's go ahead and give this a shot. And while this runs, let's bring back IntelliJ. And go ahead and start our LogProducer. So scroll back up to where you find the LogProducer and run that. Remember, now our LogProducer actually produces data directly to Kafka. Let's bring this down as well as it's producing data and bring up a browser. So go ahead and bring up a browser to localhost port 50070. And then under Utilities, Browse the file system, and you'll see a lambda directory. Under that, you'll see that Spark already created the directory for our application called weblogs-app. And it's already starting to save data under that topic, so topic=weblogs-text, KafkaPartition=0. And you can see I've actually already run this test a few days ago so there are old dates here. And there's new data coming through. So if we go in here, you'll see that our new data is actually starting to come through and get stored in parquet format. And notice that the directory structure actually matches something that you would see in a partition table on Hive. So not only will Hive understand it if you actually pointed Hive to this directory, but also Spark can understand how to read this data and do partition pruning when you're reading the data off of HDFS. At this point, feel free to go ahead and stop your LogProducer and the Spark application by simply going back to your shell and hitting Ctrl+C to stop it.

Streaming Resiliency: Demo

In this clip, we'll continue to build on the previous demo and add resiliency to our streaming application using the direct streaming approach. But first let's take a quick look at why we need to do this at all. In a normal receiving-based streaming application, the information about Kafka partition offsets is typically stored and available in ZooKeeper. So if you update your code to add features or fix a bug, as long as your Kafka consumer properties use the same group information, Spark has no problem retrieving the offset information from ZooKeeper and moving on as if nothing happened. So Spark has no problem there with cold starts or code updates. In the direct stream approach, the offset information is only stored with Spark's checkpoints. And Spark can use that on normal restarts, so there are guarantees for recovery. However, not in the case where code changes. Up to and including Spark 2. 0, code changes invalidate your checkpoints. So Spark essentially behaves like an application that never had any checkpoint information to begin with. So we need something to rely on for offset storage. You can choose to store your offsets in ZooKeeper itself. So that might be an option. However, because we're already storing data in HDFS, and Spark already knows how to create data in HDFS, then we can just rely on that. So let's bring up IntelliJ and see how that works. So we're back in IntelliJ, and I have our StreamingJob application open right where we create the direct stream for Kafka. And so what we want to do is to either use this createDirectStream as is or use a variation of the createDirectStream function call where we actually provide it the starting point. So if I go inside the function and Ctrl+P to look at the different function overrides, you'll notice that we have an option to provide a set of fromOffsets, and that is a map where the key is an instance of the TopicAndPartition class and the value is an actual Long representing the offset in the Kafka lob. So what we want to do is to create HDFS for any data, and if we receive data to actually provide it to this fromOffset parameter. And if we don't have any data, then to just use the default one, which we're currently using. So I'm going to go ahead and add some code that reads from HDFS and creates our map of TopicAndPartition and offset. So what we have here is a variable with the hdfsPath. And then we use sqlContext. read. parquet to read our HDFS parquet files. And we store that in a DataFrame, so hdfsData variable is now a DataFrame. And so now what we want to do is to get the latest offset for each topic and partition. So we use the DataFrame API to group by the topic and partition. And then we aggregate, get the max of the untilOffset, and simply call that column as untilOffsets. And so the result of this operation is still a DataFrame. But we actually want to collect these results on the driver because we actually want to use the data. So we'll call. collect on the DataFrame, which is actually an action. And because it's an action, it will actually cause the DataFrame to execute. And it returns an array of row objects. And so the result of the collect operation is that it returns the data back to the driver as an array of row objects. So if I take this Alt+= on it, you'll see that it now returns an Array of Row. So now I have a very simple Array of Row objects in memory on the driver. And I'll use the Scala map. So this is us now simply using Scala map function to iterate over the array items and not a Spark map function. So this is simply running on an Array of Row objects in memory. And so for every row, we create an instance of the TopicAndPartition class, and the TopicAndPartition simply takes a string for the topic and an integer for the partition. And so we provide that by extracting the values from the row. So we use getAs String to extract the topic and getAs int to extract the kafkaPartition. And we know that these are the column names that we've used earlier. And so we essentially get the TopicAndPartition and create an instance of TopicAndPartition. And we're actually returning a tuple of two items where the first item is the TopicAndPartition. And as you can see, there's a comma here. And for the second item, we're actually grabbing the untilOffset value, converting that to a Long, and adding 1 to it. Now we could have actually done a better job when we stored the data to take care of storing the untilOffset and fromOffset as Long values, but since we haven't, we have to return them in the same format, which was a string, and then do the conversions here. So we're converting it to a Long, and then we're simply adding 1 so that we start from the offset that follows the last one that we've stored. And then we convert the result to a map. So from offsets now is a map of TopicAndPartition and the Long representing the offset value that we're going to start from. So now instead of straight up using the createDirectStream, we will first start off by checking if we actually retrieved anything. And so we'll do a match on that. And then if it's actually empty, then we'll go ahead and apply the same logic that we did earlier. So we will create the Kafka stream without providing additional parameters. However, if it's not empty and we actually have something stored on HDFS that we can use as a starting point, then we'll create the kafkaDirectStream by using that. So even our direct stream still provides a String and String as the key and value pairs with StringDecoder for both. And then we provide the same SparkStreamingContext kafkaDirectParams. And then instead of the topic, we actually provide the fromOffsets, which carries the topic names and the partition in starting offsets for each of those topics. And then notice this override actually still has an additional parameter called a messageHandler. And the reason you have a messageHandler in this override is that you actually get more than just your key value pairs. So let's go ahead and provide an implementation for that. So in the messageHandler, we actually get a message and metadata instance. So all we're going to do is provide a simple implementation where we use the message and metadata to extract the key and the value, which is the message, and return those as a key value pair. And then, finally, for this method to finish and to work, we have to provide it with the return type of our messageHandler, and that will be the return type of the tuple of String and String, which should exactly match the return type of the normal createDirectStream. So in both cases, we're returning the exact same thing. And so now kafkaDirectStream is still an instance of InputDStream with a key and value pair of type String and String. So instead of running this and showing you how it works, there's nothing actually going to change in the output. I'll go ahead and demonstrate in Zeppelin the results of running this segment. So let's bring up Zeppelin real quick. And so you'll see that we're importing kafka. common. TopicAndPartition. Again, we have the same hdfsPath defined, and we create our hdfsData DataFrame. Apply the same groupBy, creating an aggregate where we take the max of the untilOffset, also naming that as untilOffset. And then we collect the results, which essentially runs an action and brings back the results to the driver. And then we map over the items of the array as we did earlier in IntelliJ. So let's go ahead and run this and print out the fromOffsets results. So you can see the fromOffsets that we have is actually a map where the key is a TopicAndPartition represented as this string with weblogs-text being the topic and 0 representing the partition and 48276 representing the offset that we want to start from in our next iteration.

Batch Processing from HDFS with Data Sources API: Demo

Now that we have data going to HDFS from our streaming application, let's go ahead and update our BatchJob to utilize that data instead of the normal text file that we were using. And the first thing you'll notice is that I've updated the hdfsPath and added that to our application config. and I've also modified our settings to have that hdfsPath available as a variable. And also the StreamingJob now uses the hdfsPath to read from parquet. So we'll use that same variable in our BatchJob. So let's go ahead and add a variable called weblog config (wlc), and that will be our Settings. WebLogGen. And then previously in our BatchJob we actually read from a sourceFile on the file system. So we're going to get rid of that. And because we're actually going to get back a DataFrame directly, we also don't need this input entirely. And so instead of all of this, we are simply going to use the sqlContext and then read a parquet formatted file with the location specified in our weblog config as hdfsPath. Now we obviously don't want to read the entire dataset every time, so let's add a WHERE clause. So we'll add the WHERE clause to the DataFrame. And then let's add a condition expression. So let's say we only want to run our BatchJob once every 6 hours. So we only want to grab the data for the past 6 hours. So let's take the current unix\_timestamp. So notice how in DataFrames and in Spark SQL, I can utilize any of the Hive available functions, so unix\_timestamp is one of them, and that returns the current epoch timestamp in seconds. And so we'll take the current unix\_timestamp and subtract our timestamp\_hour from that. But, remember our timestamp\_hour was actually in milliseconds, so we'll divide by 1, 000. And so let's say the result of that should be less than or equal to 60 seconds times 60, that's now in hours, and then 6 hours. And so now that satisfies our condition expression. So we already have the data and the correct data types so we actually don't need any of these transformations. And we'll simply replace the DataFrame variable here with our inputDF and register the same TempTable. And so this SELECT statement should still apply and work as expected. So SELECT product, timestamp\_hour, and the COUNT(DISTINCT) should still work, in addition to our activityByProduct calculation. In the next module, we'll start replacing this write to HDFS with a write to Cassandra and then query Cassandra in Zeppelin to look at the results from our batch calculations and compare that to our streaming calculations.

Summary

This module brings together a lot of the pieces we need for building a lambda architecture, specifically around streaming ingest and how we can use a stream to feed batch layer calculations, as well as real-time calculations. In this module, we covered Apache Kafka, the Kafka brokers, replication, availability, and the concepts behind Kafka producers and how consumers interact and are affected by the number of Kafka partitions. We then started to integrate Kafka into our application by updating our LogProducer to produce messages directly to Kafka and then building our streaming application to consume messages from Kafka. We also wanted to be able to solve the cold start problem when performing upgrades, so we used the direct stream API to extract topic, partition, and offset information from incoming data and save that to HDFS to server as a feedback loop for cold starts and also as the input to the batch processing layer essentially solving resiliency problems. With data now available in HDFS, we integrated our batch processing to pull data from HDFS in parquet format instead of reading off of flat files on the file system. So now we have the bits and pieces in place to cover streaming data ingest in batch and speed layer calculations. In the next module, we'll update our calculations to support saving results in Cassandra and serving those results from Cassandra to conclude the build-out for our reference lambda architecture. See you there!

Persisting with Cassandra

Introduction

Hi! My name is Ahmad Alkilani, and welcome to this module on Apache Cassandra. In this module, we'll introduce Apache Cassandra, a distributed database management system with unique performance and availability characteristics. We'll discuss some use cases where Cassandra can be a great fit. And we'll look at Cassandra's data model as we look at different design decisions you're going to want to make while creating your Cassandra tables. We'll also look at how we can use the Spark Cassandra Connector and use Spark DataFrames and RDDs to communicate with a Cassandra cluster as we create Cassandra tables to represent the batch and real-time views for our lambda architecture and also modify our batch and streaming jobs wrapping up this course. So without any further ado, let's get started.

Cassandra's Design

Cassandra is a distributed database management system designed for large amounts of data providing high availability with no single point of failure. So Cassandra is yet another distributed system but meant to handle storage and queries. What makes Cassandra really special is how well it scales and the speed at which it specifically can handle writes without introducing master servers or services. So no single point of failure and all nodes are equal. Cassandra also performs well for reading data. However, it has some limitations. So understanding how to model in Cassandra is critical for good read performance and because writes are so fast and Cassandra's data model typically involves writing the same data in different layouts to serve read queries efficiently. So Cassandra's typical use cases are where you have a lot of writes, web analytics type applications fit that bill, and where you know the types of queries you're going to run. So you can model your data and application for fast read performance. Cassandra isn't well suited, however, for applications requiring ad hoc queries that can't be determined beforehand unless you couple it with a search index or a Spark cluster that can take the brunt of the load for query performance. With Apache Cassandra, clients typically start up and connect to a Cassandra cluster in a similar manner to clients of Kafka. Clients typically have a list of seed nodes, essentially nodes they know are part of the cluster. This is usually a list of a few nodes and not just one so that clients can still connect to the cluster on startup as long as they can reach any of the nodes on their list. A typical production setup also won't hard code the seed nodes list in the client's config. Rather, another system that provides service discovery that is usually associated with how the nodes are deployed is used to query for available nodes and use that to connect. The important thing here is that the client only needs to connect to one node, and that could be any node in Cassandra because they can all serve client requests. Every single node in Cassandra is exactly the same so it doesn't matter to which node the client connects. So in our example, the client had two nodes, A and X. However, X is no longer part of the Cassandra cluster perhaps due to some hardware failure. But the client still has A in its list, and it can get to that node and successfully connect. Now because Cassandra knows gossip they know about each other's existence and they communicate that back to clients. And the node the client connects to essentially becomes the client's coordinator node. So any queries go through it, and results are passed back from other nodes to the client through it as well. In the case where the node the client is connected to fails, the client happily switches to another node and moves on. So, again, no single point of failure. There simply is no master node in Cassandra.

Relational Database vs. Cassaandra

Let's take a look at some of the differences in names between a relational database and Cassandra. A database in a relational database system maps to what is known as a keyspace in Cassandra. A keyspace is a lot like a database scheme if you come from an oracle background or a database if you come from a SQL Server background. It's essentially a top-level namespace or container for your schema. Replication in Cassandra is defined at the keyspace level. So one thing you want to keep in mind when grouping your tables is that they will have the replication characteristics of the keyspace they're in. That's probably one of the more important considerations when deciding if your table goes into a preexisting keyspace or if you want to create one specifically for the replication needs of that table or tables you're working on. A table in a typical database is equivalent to what Cassandra calls a column family. Similarly, a column name is equivalent to a column key in Cassandra. And a column value is the same column value in Cassandra. We'll see how these really map to Cassandra's data structure shortly.

Spark Cassandra Connector

In this module, we're going to use Spark to interact with Cassandra. And to do that, we're going to use the Spark Cassandra Connector. The Spark Cassandra Connector is a library we can use with Spark that makes working with Cassandra look as seamless as working with any other RDD or DataFrame. The Spark Cassandra Connector exposes Cassandra tables as Spark RDDs. It also comes with a customizable object mapper for mapping rows to objects of user defined classes. The library allows you to execute arbitrary SQL queries and integrates with Spark's Data Sources API for easily working with DataFrames and Spark SQL. The connector also comes with specific optimizations for Cassandra. So if you're planning on doing any serious work with it, especially when using Spark to query Cassandra, you should absolutely look at the documentation for a full list of optimizations and query performance tips. The connector offers specific optimizations for performing joins, for example, with the joinWithCassandraTable. Most of the optimizations are related to taking advantage of both Cassandra's and Spark's partitioning. For example, you can repartition an RDD to match Cassandra replication using read re-partition like Cassandra replica.

Reading Using DataFrames and Spark SQL

Let's take a look at how we can use the Spark Cassandra Connector and Spark DataFrames to query Cassandra. We'll actually cover writing data from Spark to Cassandra as part of our demos. Using Spark to read from Cassandra is as simple as using the Data Sources API providing the spark. sql. cassandra Connector class. The options required here are the keyspace and the table name. We get back a DataFrame with fields that match our Cassandra table. You could also just as easily define a temporary Spark table using Spark SQL that represents the underlying Cassandra table. So we can create a temporary table providing the class as provided to us from the Spark Cassandra Connector and define the options necessary for completing the connection. And you can also define overrides here. Then you can easily query the table just as you would with any other Spark SQL source. The Spark Cassandra Connector takes care of most of the optimizations for you and does a good job at that, for example, by automatically applying predicate pushdowns to the underlying Cassandra cluster instead of fetching everything into Spark and performing filters within Spark itself. As an aside, Spark can integrate with a Hive metastore, which is the primary store most SQL on Hadoop implementations use to define metadata about tables in a schema on read fashion. If you need more details about Hive and its metastore or Hadoop, feel free to check my Pluralsight course called SQL on Hadoop - Analyzing Big Data with Hive. So Spark uses the Hive metastore for permanent table definitions, meaning that the definition of the table persists in the Hive metastore beyond the Spark session you're in and is accessible outside the Spark session you're in. So essentially the CREATE TABLE statement persists. In the case of Cassandra and many other external sources, Hive's metastore only keeps the table definition. But Hive itself doesn't handle any of the actual data. Spark's temporary tables on the other hand as we have in this example are similar to a permanent table except that the definition only lives in the memory of your current session and is not accessible from any other Spark session or at a later point once your session terminates. This is essentially a memory-only representation of the table definition and goes away when your session terminates.

Creating Keyspace and Cassandra Tables: Demo

Now that we have a bit of understanding of how Cassandra works and how it compares to relational database, let's go ahead and prepare our keyspace and tables before we dive a little deeper into Cassandra's data model. We're back in Zeppelin at localhost port 8988, and if we go to Interpreters and scroll down a little bit, you'll notice a section for a Cassandra interpreter, and you'll notice it's identified by a %cassandra key. So that means we can use Zeppelin to run native SQL queries, and that's how we'll be interacting with Cassandra for the most part. Notice also the configuration value for cassandra. hosts. You'll want to make sure this is pointing to the correct host where Cassandra is set up. In our case, it's simply localhost, and we know that localhost from our hosts file points to our VM. So this works for us. So if you scroll all the way down to the bottom to where the Spark interpreter section is, you'll see I've added the necessary dependencies for Zeppelin and the Spark Cassandra Connector and Cassandra's core driver. Back to our notebook. Let's examine the keyspaces our Cassandra cluster already has. So we'll need to identify this section as a Cassandra CQL section by using the %cassandra keyword for zeppelin. And then we can simply select to describe keyspaces. The Cassandra Zeppelin interpreter is pretty neat in that it actually comes back with some preformatted HTML that is easy to navigate. So you can see that our cluster name is Test Cluster and that we basically only have system-related keyspaces. So let's go ahead and create a new Cassandra keyspace. So we'll add a new section, again use the %cassandra keyword because we're writing SQL here. And we'll create our keyspace called lambda. Remember a keyspace in Cassandra is somewhat equivalent to a database or schema in a traditional RDBMS. And similar to a database, there are some high-level settings that keyspace and Cassandra have, for example. Here we specify the replication strategy and replication factor for all the data within this keyspace. Again, remember tables within a keyspace are not only logically relevant but also share the keyspace's replication strategy and settings. We're using a replication factor of 1 here because our VM only has a single Cassandra node. Not very useful in a production scenario of course, but it'll do for our demo purposes. Let's go ahead and also create the tables we're going to use. Remember, CQL is very similar to normal SQL. But it has its own distinct features. We'll quickly walk through what we've done here and then go into Cassandra's data model in more depth before we start sending data to these tables. So we're creating four new tables under the lambda keyspace. In reality, we have two distinct tables, one for activity\_by\_product and one for visitors\_by\_product. And we simply duplicate these, once for streaming and once for batch. The idea is that the results of our batch queries go into separate tables and the results of streaming aggregates go into their own tables as well. When presenting the data, you can combine the results from both tables to present a global view, one with accurate results but high latency from batch processing, and another with a low latency view and potentially less accurate view of the data coming from your streaming layer. Our tables are actually pretty simple here with a product of type text, and all other fields have a type of bigint. The more interesting bits, however, are towards the end of the table definitions. We have a definition of a primary key, which consists of two columns, and also a clustering order by clause. The general idea here is that Cassandra similar to other database systems allows for a primary key, and composite primary keys like this are very common in Cassandra because the first segment of the key actually represents something more important called a partition key. That's the key Cassandra uses to partition the data. And then the data is further sorted by and grouped by the remaining columns, so in this case timestamp\_hour. You can also override the ordering for those columns providing a clustering order by clause and defining the order. This is especially useful for performance if you know you're primarily interested in data based on latest values of timestamp\_hour. So, again, the first part of the primary key is called a partition key, and the remaining columns of any composite primary key in Cassandra are called clustering columns. Don't fret if this looks a bit too complicated. This is exactly what we're going to examine in detail in the next section about Cassandra data modelling. But I wanted to give you a taste of what we're going to be using so you can relate to these examples. And one last thing before we move on to Cassandra data modelling is that Cassandra always operates in upsert mode for rows. Upsert stands for update and insert. So that means if there's a match on the primary key, Cassandra will always update the row. And if there's no match, then it's an insert operation. You can control how updates are performed for complex collection column data types like maps, but that's beyond the scope of this course. We're actually going to take advantage of the fact that Cassandra always issues update/insert type operations or upserts for short as we continue to stream updates to the data from our StreamingJob. Imagine that we will send an update every minute or every few seconds depending on the window interval with the latest information from our streaming state snapshot. So our streaming state snapshot will update any previously written data from a previous batch or window. And Cassandra treats every insert operation as an upsert, and that makes Cassandra ideal for this type of operation. Next up, data modelling in Cassandra.

Data Modeling with Cassandra: Part 1

Now that we understand a little bit of terminology around Cassandra, we'll take a look a Cassandra's data model. The first thing to note about Cassandra's data model is that it's a row-store, so it's not another column store aimed at handling massive analytic type queries. It aims to solve the high volume write problem with excellent performance for single row lookups or contiguous row lookup for a single partition key. Cassandra still gives you a table representation and query language that is similar to SQL called the Cassandra Query Language, or CQL for short. It's very similar to SQL but comes with features specific to Cassandra while also omitting a lot of the SQL features that Cassandra simply wasn't built to handle like joins for example. To better understand how to model your data and query Cassandra effectively, it's good to get a mental map of what Cassandra structure really looks like. You can think of a Cassandra cluster as a glorified distributed map. As in any map or HashMap, you can access its member by key. And Cassandra's no different. Access is by key, and in fact, this is so important to Cassandra that the key is given a special name and is referred to as the partition key. So your top-level data structure is a map, and Cassandra uses a partitioner to produce a consistent hash for your key values. Cassandra takes a token range for all of the possible values for a hash and divvies that up between all of the nodes in the cluster so each node gets a contiguous range of token values. As nodes are added and removed from the cluster, this token range is split or merged, and all the nodes have a little table in memory where they keep track of which node is associated to which range of contiguous tokens. They all end up having this information because they all gossip about what's going on in the cluster. So they all end up with the same table that tells them who's responsible for which range of tokens. On write, the key is hashed producing a token, and the coordinator node knows where to send the data because it has the table of token to node mapping. Pretty simple in concept. A similar thing happens on read. Cassandra also introduces a concept called virtual nodes where each node has a set of virtual nodes within it. We're not going to get into this too much. However, the general idea is to help with the data movement when the range of nodes expands or contracts. The original concept behind the simplicity of the token to node assignment is under the assumption that these token ranges were contiguous. The virtual node idea splits a node's range into multiple smaller ranges each of which is a contiguous range but each covers a different portion of the total range. This essentially allows nodes to copy data for more neighbors for replication and recovery purposes. Again, we're not going to cover everything Cassandra has to offer, so if you want to learn more about virtual nodes or other concepts we don't cover in this course, Pluralsight has a course on Cassandra that you can check out.

Data Modeling with Cassandra: Part 2

So far we've explained how Cassandra's top-level data structure is a map, but it actually gets more interesting as we uncover how columns and column values are stored. Cassandra's data structure is actually a distributed map where the first key represents the partition key as we've already seen. And it's the partition key that determines which node the data for that row goes to using a partitioner. The value for each row is actually another nested map or, rather, a sorted map where the key of the sorted map is the column name or the column key and the value is the column value. And this sorted map is sorted by the column key. Let's take a practical example. If we represent our commentsByUser table as this map of sorted map, the username in our example was our primary key, and the primary key also defines the partition key. The primary key, however, doesn't have a 1 to 1 mapping with the partition key because a primary key can be a composite key similar to most databases. In this case, the primary key has a composite key meaning it consists of more than one column, then the first column defined in the primary key is what defines the partition key. So, for example, if we defined our primary key as a combination of both the email and username where the email was the first column, then the partition key in this case would have been the email instead of the username as in our previous example. So the very first column that shows up in a primary key is called the partition key, and the partition key has its name because it's used to define the partition that data goes to. So what about the remaining columns of the primary key. Well, columns 2 onward are called clustering keys. So username in this case is a clustering key. Clustering key values are used as part of the sorted map column key so data ends up being sorted by clustering keys in their order. This also means you can use clustering keys in your queries as predicates. It's critical to understand how the partition key is defined because Cassandra considers partitions as the smallest unit of atomic storage. And you can't look up values in Cassandra without providing the partition key in your query. No partition key, no query. So primary key also defines the partition key. Good! We now know that. In our example, we also had two columns, comment and email. So these are physically represented as the keys in the sorted map, and the column values are the values for for the sorted map. Notice how this structure opens up the door for some specific optimizations on one hand and limitations on the other. One limitation in Cassandra is that queries are very strict. Any query must at least provide a partition key. So any WHERE clause you have, in fact, any query in Cassandra must include the partition key. Remember, that's the first column in your primary key in case of composite primary keys, and it's the only column in your primary key in case you only have a single primary key column. Let's take a look at another concrete example and discuss composite primary keys.

Composite Keys in Cassandra

In this example, we're modelling a table for ratings of movies by year. We're interested in capturing ratings from IMDb and Rotten Tomatoes, two popular sites for moving ratings. Notice again how the table name is an indication of what you can use to query the table and what data it provides. So from this table name, we can query this table by year and movie name. Looking at the primary key, we're using a composite primary key, which means the first component is the partition key so data is partitioned by year. And the second component is the movie name, which means we can further look up an item within a year by its movie name, and the movie name is a clustered column. Looking at the physical storage model, you'll see how a clustering column's values actually become part of the column key and are set up in the order specified by the CLUSTERING ORDER BY clause. So clustering columns essentially divvy up a partition into what you would see as a row in the Cassandra query language or equivalent relational table. Now this is actually a pretty bad example of how to create a partition key because data for a given year will only go into a single node based on the year value so all the rows for a year will go to a single node if we use the setup. That not only means that that single node would need to have enough capacity to handle all the data, but also impacts performance if the most recent year is where most queries would go to. Most other nodes will remain idle. So the year alone is a bad partition key. However, we can introduce a composite partition key. Let's see how that looks. So now we have a new column called hash\_prefix, which can be created by taking the hash of a column, for example, the movie name, and use that value modulo a number for the number of partitions we'd like to split a given year for. In this example, we only have two movies, both in year 2014. However, their hash\_prefix ends up with a different value, 2 in the case of Birdman and 3 for Interstellar. So they land on different partitions and potentially different nodes depending on the actual murmer 3 hash value produced for the year and hash\_prefix combination.

Modeling Time Series Data with Cassandra

An interesting observation results from the fact that the sorted map's keys represent the column names. That means the list of columns for every partition key don't need to be the same. That sounds confusing, doesn't it? But Cassandra indeed even through CQL allows you to define a table where the number of columns for each partition are dynamic. Because the second map is a sorted map, you'll see data models for modelling time series data. Take advantage of that where the sorted map's keys are actually a timestamp so the column names are timestamps and the column values are values for that timestamp. The fact that it's a map allows for that, and the fact that it's sorted makes this efficient for range queries. Cassandra can handle billions of columns in a so-called wide-row table. As we've seen earlier, we can even use composite keys for the column key or row key. Here's an example where we have a composite key that consists of event type and timestamp. Notice how data is sorted by the first column in the key. And as a result, range and lookup queries have to have that column. Cassandra's all about denormalizing your data to serve specific queries so you'll see a lot of tables that are called specifically to match the queries they support. In this example, we can't query on the timestamp range without specifying the event type we're looking for. So the fact I ordered my composite key this way means I know the event type is of higher importance or rather that it will be used in queries that access the table. We can flip the timestamp and event type column order in the composite key, and then the table will be efficient to query by time range. We'll practice creating tables in CQL in our demos so I won't be covering that in slides.

Spark Streaming Realtime Cassandra Views: Demo

In this demo, we're going to adjust the last bit of our code to save data out to the Cassandra tables we've created for stream and batch processing. The first thing we need to take care of is to make sure you have the correct dependencies for Maven to pull the Spark Cassandra Connector and the Cassandra driver with their appropriate versions to match our Spark, scale, and Cassandra setup. Now that we have that taken care of, we need to first tell Spark where to find our Cassandra cluster. So I've modified our SparkUtils and the getSparkContext function to add a configuration property so the Spark Cassandra Connector knows where to go. So that's all you need to get ready. And I'll make our StreamingJob full screen to get more real estate. Now there's a slight difference in approach that we'll take between our StreamingJob and our BatchJob. In our StreamingJob, we'll use the connector's streaming implicit conversions by importing com. datastax. spark. connector and com. datastax. spark. connector. streaming. So what we want to do in our StreamingJob is that we want to take the results of these state snapshots that we've created in module 4 and instead of just creating a table out of the state snapshot and showing the results of that table in Zeppelin, we want to save those results in Cassandra. So let's see how that looks now. So I'll scroll down to where we previously had our stateful aggregations for ActivityStateSnapshot. You'll see towards the end we had a foreachRDD after the reduceByKeyAndWindow. And we essentially mapped the results to an ActivityByProduct case class and used that to create a DataFrame and register a TempTable that we queried to see the results of each window in Zeppelin. Well, it turns out that the Spark Cassandra Connector has different approaches you can take to save RDD data to Cassandra, and one of the easiest and less error-prone approaches is if you have a Scala case class like we do here with column names that map directly to the destination table. The Spark Cassandra Connector allows you for a little bit of leeway in the column names. For example, the connector will understand how to convert Scala CamelCase properties to a Cassandra column in snake\_case, so essentially with underscores. And I encourage you to check out more documentation on the Cassandra Connector mapping to see all of the possibilities. In our case, we simply have an exact match between the column names, and that also works. So instead of this. toDF to create a DataFrame, now that we have the implicit conversions from the Cassandra Connector, we can simply replace that with a. saveToCassandra providing the keyspace as lambda and the name of the table we're saving to, and that's really all it takes. In fact, we can optimize this a little bit more. The Spark Cassandra Connector also has implicit conversions that provide a DStream with a saveToCassandra method. So we actually don't have to drop down to the RDD to use the saveToCassandra implicit conversion method on the RDD. It'll also exist on the DStream. So we'll completely get rid of the foreachRDD here. We'll move the map statement to a map on the DStream directly. Remember, DStreams actually implement similar but not all of the transformations that are available on RDDs. So a simple map on the DStream here gets us a DStream of activityByProduct. And then we can simply run a saveToCassandra against a DStream. You'll see I've also done the exact same thing for the visitorStateSnapshot going to the stream VisitorsByProduct table in the lambda keyspace. So now that we have our StreamingJob set up, let's go ahead and build our project to test this out. So we'll package our application, and I'll fast forward as it compiles. And then we'll copy the shaded uber jar from the target directory, place that under the Vagrant images shared directory (remember, this is where you cloned the Git project), and then under the Vagrant folder. I'll also bring up Cygwin, and I'm already inside of the VM and under the /Pluralsight/spark directory. We've done this a few times already, so you can check out our previous clip if you want these step-by-step instructions. Before we run our job with spark-submit and start producing data using our LogProducer, remember that Spark Streaming doesn't handle code changes and checkpoints very well. So if you know your code has changed, then it's best to blow away the spark/checkpoint directory from HDFS. The VM I've created for you already comes equipped with an HDFS client, so we can simply run hdfs dfs -rm -r and -f so that we delete everything under checkpoints recursively including subdirectories. And we can skipTrash so we skip having this data going to trash. However, in production, you may want to make sure that you take a backup or snapshot of this checkpoint directory. If you ever want to come back to it and run your old code from where you left off, you'll need this checkpoint. Another thing I like to delete for demo purposes and specifically because we haven't handled this scenario properly in our code is that I like to delete the data we've saved in HDFS. If you remember the code we've implemented to handle code upgrades and restarts looks at the offsets in HDFS, and it attempts to start the direct Kafka stream using those offsets. However, in practice, while you're developing and testing and especially as you're learning, there's a good chance that Kafka has already rolled off those offsets from the topic. So if you do have a scenario where you've left this for a few days and end up seeing an offset out of range exception, it's because you'd be trying to run your code that starts from an offset in HDFS that no longer exists in Kafka. This means your consumers were offline for too long, and the data you're looking for no longer exists in Kafka. Remember, Kafka doesn't keep everything. It actually rolls it off. So essentially the smallest offset in Kafka is bigger than the one you're looking for. This is a scenario you should handle in your streaming code by creating the Kafka brokers first. But because we haven't done that and I want to avoid seeing that exception, I'm going to go ahead and simulate a fresh start by deleting any previous data we had in HDFS. So now similar to a demo in module 5, we're going to launch our Spark job locally using spark-submit. And then we'll head back to the IDE to run the LogProducer so that we get new data. We can also bring up Zeppelin again and see if we're getting data in our Cassandra tables. Sure enough, we start to see data trickle in from our StreamingJob for both activity\_by\_product and visitors\_by\_product. So now an application that is interested in a near real-time low latency view of the data can look at these tables and present the data as fresh as it's being produced, and we obviously have controls over that with the length of the window in the streaming application. So that essentially takes care of our StreamingJob. We'll handle the batch layer next.

Spark Batch Cassandra Views: Demo

So for our BatchJob, we'll take a slightly different approach. You'll notice we're already dealing with the DataFrame because we're using Spark SQL to run our queries, so visitorsByProduct is a DataFrame. There is no saveToCassandra implicit conversion method on a DataFrame provided by the Cassandra Connector. However, because we have a DataFrame, we can use the generic Spark Data Resources API that the Spark Cassandra Connector also provides classes for. So we can simply take the DataFrame, call. write, specify the org. apache. spark. sql. cassandra class as the format, and provide options that tell the class which keyspace and table we're saving to, and then call save. And we can do exactly the same thing for activityByProduct. I'll save you the time for building the project, and we'll run this in Zeppelin instead. So I have the same batch code set up here in Zeppelin. And because we've been running our StreamingJob recently, I know we have data in HDFS, so this should render data that goes into our Cassandra batch tables. I'll run this segment and fast forward the video here a little bit. So that ran successfully, and we can use CQL to create the data we now have in Cassandra.

Summary

In this module, we demonstrated some of Apache Cassandra's strengths, for example, how Cassandra can be an ideal solution for write-heavy workloads while taking special care to model your tables to support the queries you expect to have. Cassandra also encouraged duplication of data where each table is tailor-fit to handle specific queries effectively. And because it's so good at handling writes, it can deal with the load duplicates add to it. We also note in this module how Cassandra isn't meant to deal with ad hoc or analytical queries, but Spark running side by side with Cassandra, perhaps even in a virtual data center, can help address those concerns. As we wrap up our work on the lambda architecture, we looked at how we can create similar tables to represent the batch and real-time views as a result of the batch and speed layer computations of our lambda architecture. And we also looked at the Spark Cassandra Connector to help with the integration between Spark and Spark Streaming with the Cassandra cluster. You'll still obviously have to build in some smarts in your code to determine when to look at data from each layer, and that's something you'll need to consider in your presentation layer where you display results. So this wraps up this course. I hope you enjoyed it. And a final reminder: Don't forget to check out the GitHub page for any updates or troubleshooting tips and tricks and especially ones related to the VM.

Course author

Author: Ahmad Alkilani

Ahmad Alkilani

Ahmad Alkilani is a Data Architect specializing in the implementation of high-performance data warehouses and BI systems. Ahmad enjoys over 10 years of broad IT experience including SQL Server,...

Course info

Level

Beginner

Rating

4.7 stars with 104 raters(104)

My rating

null stars

Duration

6h 4m

Released

15 Nov 2016

Share course

Features

·

Authors

·

Mobile & offline apps

·

Blog

·

Help center

·

How-to videos

·

Referral program

·

Terms of use

·

Privacy policy

Send feedback

Get support