Video Transcripts

Module 19: Processing Big Data with Spark and Airflow

Video 1: Introduction (02:10)

The big data and distributed computation platforms broke new ground and gave us the capacity to deal with the volume, variety, and velocity that is typical of big data. However, there were a number of shortcomings as well, and that's usual when it comes to first generations of any type of idea. We needed something that was easier, that was simpler, and that was faster, and that brought about a new generation of analytical engines. Now, thankfully, these are all open source, which means that we can get access to the full source code, and we can modify them and run them without having to pay for licenses. Now, this is a great one to know and understand the adoption in industry is wide, and it is one that has a very active community. So, this is the one that we'll be taking a look next. If you've been in computation for long, you're familiar with trying to create some scripts to be able to automate some tasks that you're trying to perform, to trying to shortcut the number of manual steps that you need to take.

Now oftentimes, some of these become ones that our organizations are dependent on, which means that they need to go through a maturity curve. There's many cases you didn't consider when you were simply running on your machine. And some of these might be failures, monitoring, conditional dependencies, scale, deployment, history, we could go through a long painful list really here of what those are. Now, thankfully there are some platforms that look precisely at this type of scenarios. They're called workflow engines. And in this case, we're going to be looking at one that was born in the data engineering world, which means that was looking at a lot of the pipelines that data engineers were building. And that is a great fit for many projects, that it is widely used in the community, and it is one that is very helpful to know. So, let's go ahead and jump in, and take a look at a workflow engine.

Video 2: The Evolution of Big Data Architecture and an Introduction to Spark (07:06)

Prior to the web, the generation of data was quite different. You have pretty structured

generation that could fit neatly, or at least most of it, into rows and columns. It was a world where you had low velocity, variety, and volume. However, once the web comes along, we really have an explosion of data sources, some of which are still being added today. We certainly had more velocity and more volume, and that didn't fit neatly into that vision of the world that existed prior to it, that world where all of the data went into tables, into rows, and columns and relationships between these tables.

What we had was a big data problem, one where we had velocity or high velocity, variety, and, volume. And companies at the time we're running into this problem, running into this problem of how do we store this tremendous amount of data, this variety and these, this velocity that we have. And so Google in, in the time broke new ground, introduced new innovations with three papers that are widely cited. The very first one is the GFS, the Google file system where they could distribute this oncoming flood of information onto a fabric of commodity servers. So, leveraging low cost hardware, being able to overlay a fabric that could address the diversity, they're the failures that could occur and provide a robust system. On top of that, they added MapReduce, a programming model, and there were other papers that are widely cited such as big tables and others.

Now that was a proprietary system, but the publications detailed well the work that had been done, and so the community was able to leverage that and build an open source version of these ideas. The very first layer was called the Hadoop Distributed File System. The second layer was called the same thing, MapReduce. And on top of that, you had a resource negotiator. Now as you can imagine, this was a great step forward, it broke ground, it changed the capabilities, it provided a tremendous resource that was open source. You didn't have to pay for licenses, you could leverage it. It broke away as well from the cookie cutter solutions of data warehouses. However, as tends to happen, once you take a big step forward, you find you need additional things and so there were a number of challenges. One of them was this was open source hardware. It was hard to maintain, it was hard to version, it was hard to evolve. The other thing as well is that although MapReduce was a great programming model, it wasn't necessarily what all industry and certainly verticals were used to, especially within the data community.

Another problem as well was that a speed, reading and writing intermediate results to disk, as is often done in the MapReduce model, was one where you had, depending on the type of loads and work that you were doing, pretty slow performance. This reading to disk versus writing to

memory is a tremendous difference when it comes to speed. It's like going down to your gas station or going all the way to the moon or further. And so as you can see here, I'm going to expand a little bit more on why MapReduce was not enough. Ultimately, there were a lot of collisions with the knowledge that already existed in organization, the practices that had already established. And in this case, I'm referring here to the very first bullet, that of SQL, where that had been adopted by the data industry or the data practice as the model when it came to querying data. And so asking them to do MapReduce was really difficult when it came to getting those communities to buy in.

Another is that machine learning wasn't integrated in a way in which it made it convenient to use. And so as you can see there, the Hadoop community adds a number of packages there, Hive, Storm, Impala, and so on, that address many of these other needs. However, the result is that you have even more complexity, even more difficulty in being able to simply install these packages and get a running system. And so there's really a need for a new platform. And that is precisely what Spark focuses on, something that is easier, provides more interfaces, more programming model, it's simpler. It is only one platform now, this multitude of pieces that you have to bring together and it is faster. It writes those intermediate results to memory instead of to disk. More than that, it also decouples storage and compute.

So, you can see here from this illustration, you don't only have to go to Hadoop, as you can see there, the little elephant, you can also go to many additional data sources. And so this turns out to be a very nice architectural abstraction to be able to work with the ecosystem of sources that may be available. And so ultimately they achieve very much that goal. As you can see here, you have a number of different packages for different types of workloads. You have SQL, you have data frames, which is rapidly emerging as one of the default go to structures when it comes to Python in that community, you have streaming data that you can leverage as well. A lot of efforts go to that especially as you near or your approach real time, being able to leverage the data real time. You have machine learning algorithms and libraries built into the platform as well and even graph processing.

Now all of that is on top of an engine as you can see there, and it is possible to program against this platform with multitude of languages. There you can see SQL, one of the most important ones, you can see Python certainly in the data community that is key today, as well as some other ones that you may want to use such as R and Java. And so as you can see here, Spark really

addresses the need of a new generation of data needs of big data, whereas Hadoop really was a big step forward, certainly a stepping stone, a building block that continues to be used today. However, Apache has really brought that generation forward and created really a very performant, very fast, easy to use new platform that has made a tremendous impact in the big data world.

Video 3: Creating Spark Docker Images Exercise (03:12)

When it comes to installing platforms, there's no substitute for Docker. They have really addressed the problem of being able to run somebody else's code with a beautiful abstraction. Now, when it comes to Docker itself, you could build your container from the ground up and create something that is reproducible, that is certainly lean and trim and fits your needs to perfection, or you could use one that the community has created. Now, being able to create that installer in Docker could take some trial and error. And for example, one of the things in this specific platform that you'd want to be able to do is you'd want to be able to package onto your container, both the Java development environment as well as the Python development environment, plus all of the dependencies that might be required for Spark itself.

So, as you can imagine, that takes some effort. And because of it, it is often easier to be able to identify a Docker container or a Docker specification that the community has produced and that we can leverage. There are a couple here that I will mention and I'll start off by talking about the one from Big Data Europe. And if we navigate to the project, you can see here that this is a community that is devoted to providing resources for researchers, scientists, and others that might be, that might be interested in posing big data type questions. If we navigate to their Spark repository, we'd see their Spark specification. And you can see if you download, if you scroll down the page that you can see here the Docker compose file. And if you read through their document, you'd see how they expect you to use it. Now, there's an additional image as well, and this one is by Bitnami. And if we navigate to Docker Hub to their address, we'd be able to see here the image that they have created.

As you can see there as well, that has been pulled for over million users. Now, if we scroll down the page, you can see there the instructions. And after having looked at both projects, this one is much simpler to use. You can see there that you simply pull the file curl in the instructions, and then you can do Docker compose up. Now, at this point, go ahead and pull the file and save it in

your local machine. If you're interested in knowing more about how that was put together, you can navigate to their repository and read more about that image. You can see here what is Spark, the short of it, the why use Bitnami images and so on. And as mentioned initially, you can always trim one of the existing ones or build one of your own from the ground up. So, once again, go ahead and pull the docker compose file and save it locally.

Video 4: Creating Spark Docker Containers (02:06)

It is now time to create our containers, so let's go ahead, and move to the command line and do so. I'm inside of a local directory that I have called bitnami. And if I look at my docker compose file, you can see there that I have the contents listed. I'm going to go ahead and clear the screen, and now I'm going to go ahead and enter docker-compose up. Now, before I do, let me go ahead and show you that I have a clean machine, meaning I don't have any containers running and in a moment, I will be installing them. However, even though I don't have any containers running, I do have the images on my machine from previous installations. So, that means that when I run docker-compose up, I will not have to pull those images, so my installation will be faster than yours.

And so as you can see there, we have installed Spark. And if I list my containers, you can see there that I now have three. You can see there that the very first one is Spark, and then we have two workers with it as well. So, before you go forward, and make sure that you can see the containers. And there's one more thing as well, you can view your current status by navigating to your local host, and then the port 8080. So, let's go ahead and do that now, localhost 8080. And as you can see there, we get a current status from the application. We have a couple of workers. We have no applications running or completed at this point. In other words, we have a clean installation. So, that's your turn. Make sure you can walk through that installation and that you can confirm that you have all of your components.

Video 5: Loading Flight Data into the Spark Docker Container (02:49)

Now, that we have a Spark system up and running, let's go ahead and upload some data into that container, and then inside of that container, we'll go ahead and start the Python shell within Spark. The general pattern to be able to copy data into the container is the following. We use the Docker command, CP, to copy, then the file name, then the container name, then the file name. Now, as you can see here, I am inside of a directory called data. And if I list my files, I

have the file that I'm going to be uploading and that is a file that's called departuredelays.csv and has my over one million rows of data. Now, in order to be able to target the container, I need to know the name of it. And as you can see there, the one I want to target is Bitnami_spark_1. So, let's go back to our command line and I'm going to paste here my command docker copy departure delays, the name of the container, and then the name of the file. So, let's go ahead and hit return on that. And you can see there that there's a bit of a pause while that is uploaded into the container.

So, now that that is in place, let's go ahead and access the container. And we can do that by navigating to the container and accessing the command line. And as you can see there, I have one more window that is just opened and let's list our files. And as you can see there, I do not see it. Let's go ahead and check our path. And yes, I uploaded it to the root. And as you can see there, let's see if I am now in the root and I don't see it yet. There it is, I've missed it previously, but it is here, departuredelays.csv. Now, the last thing that we want to do is that we want to confirm that the system is up and running, which it is since we just logged into it. But we want to make sure that we want that we can run the Python shell. So, let's go ahead and do that. I'll Now, go ahead and enter pyspark. And as you can see there, we have the shell up and running. So, now it is your turn. Make sure you can go through the steps that we just walked through, make sure you can upload your data, and make sure that you can start the shell successfully.

Video 6: Using PySpark to Query Flight Data (07:29)

Now that we have a Spark system up and running that we have some data, let's go ahead and do some analysis of that data using SQL, one of the languages that is supported by Spark. We'll start off by creating a Spark session and I'll give you a walkthrough through the steps that we're going to be doing, and then later we'll come back and write the code in detail. The very first thing that we'll do is that we'll read the data and we'll infer some of the schema, we'll create a table as well, we'll look for the flights that are greater than 1000 miles as an example. We'll then look at some of the delays. This is one example. Here is another one where we're trying to see how long of a delay we have. So, as mentioned, this is just a quick overview.

Let's go back and walk through that code one by one, and we'll start off by creating a session. So, as you can see here, I am at the console, on the Spark shell, the Python Spark shell. I'm going to go ahead and clear my screen here, and I'll get started here by creating a session as you can

see there. Well, at this point, I'm simply importing the package. Next, I'm going to go ahead and create that session, I'm going to name it Departuredelays. Next, I'm going to go ahead and create a variable that will hold the path to our data, the departuredelays.csv file, the file that has over a million rows. Next, we're going to go ahead and read the data and as you can see here, there's a few points worth noting. One that the format we're telling it to expect CSV, another option there is that we are inferring a schema. That is, take a look at the file, make your best guess, and as you can see there, that will be good enough for what we're trying to do, this is a pretty simple file structure.

And, so let's go ahead and execute that, and as you can see there, that takes a moment. We have quite a bit of data in that file, but you can see there that, that now has gone through. Now, we're going to then, based on that, we're going to go ahead and read that into a view. As you can see here, you can think of it as a table. So, now that we have a session, that we've loaded the data and that we have a table, we can go ahead and start to write some queries. And as you can see there, that is the SQL syntax that you would expect. We're selecting and we're specifying there the columns that we're looking for, the distance, origin, destination from delays table. And if we go back, you can see here where we specified that table, that view, we gave it a name. And then, where the distance is greater than 1000, we're then going to order by, we're going to do an aggregation and you can see there by distance and descending order. We're simply going to show, we're only going to show 10 there.

So, let's go ahead and hit 'return'. And as you can see there, we get a result nicely formatted onto the console. So, it's worth noting here what took place. We uploaded a CSV file. We, then, infer the schema from it. That is, we do not have to go through the usual exercise of being able to, of defining tables, and creating data types and so on. And then, we were able to write a query in SQL against that data. So, you can see that there, that's pretty speedy, pretty fast to be able to do this. Now next, let's say we were interested in flights that had delays of over two hours between a couple of well known airports, that is the San Francisco airport and the Chicago airport. So, let's go ahead and enter another query. And as you can see there, we're selecting for date, delay, origin, and destination from the delays table where delays are greater than 120 and the origin is San Francisco and the destination is Chicago. We're going to order once again and let's go ahead and return. So, as you can see there, we get the delays, we get the origin and destination, and we get the dates as well. So, once again, pretty handy functionality, great

combination there of SQL, Python, and the platform of Spark.

Now, one more in this case, we want to get a sense of the types of delays that we have. And, I've gone ahead and pasted the query onto the command there as onto the console. You can see there that we're selecting, once again, the delay, the origin, and the destination. But in this case, we're looking at a case statement and we are going to report on very long delays, long delays, short delays, tolerable delays, and so on to be able to get a sense of what this data looks like. Now initially, I'll run it quickly just this is there and you can see there the data that is coming back. But, let's go ahead and update that so that we can see more of the data. Now as we do that, you can see here that we start to get tolerable. We get some earliest, some short and some long. So, you can start to get there more of a sense of the data. It's easier to parse, certainly visually, and you can see, there the type of statement that we wrote.

Now, to illustrate the flexibility, we have been working on the traditional SQL model, but to illustrate that we can do the same with a more traditional use of a data frame. Let's go ahead and import that package and then write the very first query that we did, but do it using the dataframe. And as you can see there, we can select and we can specify there the columns and then you can do the filtering. As you can see there, the WHERE clause and then the ORDER BY. And if we hit 'return', you can see there that we get the same output that we originally got when we did the SQL query, which I'll repeat here again just so we can have it next to each other. Well, I wasn't quite able to get it next to each other, but I think you see there the comparison of both. So, now it's your turn. You can flex your SQL muscles, go ahead and dig into this data. This is a fun dataset. You can post many more questions and you can explore here within the shell what you can do.

Video 7: Airflow: A Workflow Management Platform (06:57)

When it comes to data platforms, there are many sophisticated arguments that are made by vendors of their platforms. However, when it comes to airflow, this is a pretty accessible pitch. What they're looking at, what they're focusing on is a workflow management platform. And what is a workflow in this case? It's pretty simple, pretty much what it sounds like. A sequence of tasks, something that might launch on a schedule or perhaps on an event, something that is used to build data pipelines. But as you can see, this is really a pretty general problem that applies to many things. So, let's consider what one of these might look like and I'll look here at

pulling data, say from transit systems in a city, something like buses, trains, or perhaps ferries. And as you can see here, you start off with raw data and ultimately you have some report or intelligence. You might want to pull that data, do a transformation perhaps to a DOM, extract some tags, from those tags, mine that data, pull out the factors you're looking for and can do a visualization or a report.

Now, there's many ways you could do this. And this is what many of us have done at one point of another over the years, written what's often called glue code, to be able to bring together the different parts that might do that pipeline. You might start off with a bash script that runs on Python to initially pull that data. Then, you might want to do some cleaning, you could do it with Python or some other language. You'd put it into a DOM, if you had some HTML type data, you'd run some JavaScript over that. You'd then pull some tags. You'd do some analytics, perhaps go to a database, use some of the support that is given there, and then ultimately create your visualization or your report. Now to be able to run this periodically, you'd put it into a Chrome job, and over time you'd hit all of those edge cases, all of those problems, and you'd mature and grow it, but it wouldn't come without pain.

There are a lot of common problems that you might run into. One of them would be failures. What if it fails? Do you immediately try again? Do you wait? How long do you wait? On platforms that are URL endpoints, you might be confused or that you might be mistaken for a denial of service. So, you need to be careful with that. You need to be able to monitor to capture the amount of times that you have failed, that you have succeeded. You'd eventually want to be able to run some analytics and see how you're doing. There would be also conditional dependencies. What order do you need these tasks to execute? And what if you had a failure? What if you're considering cases where you might get different results by different tasks and you want to do some branching based on that? It also might want to exceed one machine. You might want to have a number of these performing these steps. How do you coordinate between them? You might want to have some deployment considerations, some versioning of your code, and ultimately you might want to rerun some of these historical runs through some of these pipelines.

So, as you can see there, that's a lot of things that are eventually captured by a lot of the homegrown type scripts, but that are challenging to do. And so when it comes to Airflow, this is very much the situation the original author was considering. And you can see there, that a lot of

what I've just pointed out was captured on this platform. This is a framework built for that from the ground up. It defines tasks in Python, something that is frequently used in the data community. Something that has the capability to spread throughout different nodes. Something that can capture the history, that has a graphical user interface that allows you to immediately see how this has run, how mature, how successful it uses colors as we will see, that it has an extensible plugin mechanism to a lot of the infrastructure that you'd want to connect.

This is something we didn't even mention, the cycle of trying to find all of the different drivers that you need, the adapters that you need for the different types of databases, and something that has struck an important need when it comes to the community and that has gained wide adoption. Now, the model that it uses to define these tabs or these pipelines are directed acyclic graphs, often called DAGs. And as you can see here, that is a pretty good fit for tasks and the type of sequences that you might want to run at certain points. You might want to have a multitude of these. You might want to have some branching depending on the results of certain intermediate steps. So, why Airflow? For many of the reasons we have mentioned, but I think, ultimately, it comes, to you need it anyway. You need something like it anyway. And so, why not right on top of something that has been built to capture many of the use cases that you're going to run into. And so, this is saving you a great deal of war.

Now, the common use cases are here, I'm listing a few, we've already mentioned some, but ultimately warehouse is something that uses this type of patterns quite a bit, the ETL, extract, transform, and load. And, when it comes to machine learning and data science, there's even more of those you can see there and a few others that I'm listing. And, you can see there that I'm also pointing to a URL when you can learn more about this. But if we talk about data science and we talk about the data hierarchy of needs, you can see there that the bottom of the pyramid, the collection, the movement, the transformations are great fits for Airflow.

Now, the rest of the pyramid could make use of it as well. If we were to look at the data science cycle of going from raw data to deployment with data preparation and training in between, you can see there that you're using a multitude of packages. Now, when it comes to the glue that goes between these, you can start to, you could try to do it by hand, as we mentioned before. You could try to use some package that automates a lot of this, but ultimately they're not going to have support for everything that you have. Or you could start to use Airflow. You can see there the different packages and being able to provide that glue might be a pretty useful thing. So, in

general, this is something that we need to build anyway and so having a workflow management platform is something that is pretty useful.

Video 8: Creating a Workflow in Airflow: Directed Acyclic Graphs (DAGs) (03:56)

Writing a workflow in Airflow is very similar to simply writing a Python program. In fact, what you end up is a Python file. Typically, you do your imports, then you do a number of tasks, and then you clean or close somehow. Now, when it comes to writing workflows for Python, you have that Python file that I mentioned, but you need to give it some structure so Airflow knows what you're trying to do. You start off with the imports as you usually do. Then, you'd have your DAG arguments that you're going to pass into your DAG instance, your tasks, and some way of expressing your dependencies. So, let's take a look at each one of these in turn. We'll start off by looking at the imports. You'd start off by pulling in Airflow, as you tend to do whenever you're working with a new package. Then, you would define your operators, we will need these when we define our tasks. You can see there that we're going to be using a bash operator and a Python operator.

Now moving forward, we have our default arguments. As you can see there, many of these map to the usual concerns that you have whenever you're writing a workflow using scripts, using glue code from the ground up. You have your owners who you notify in case of a problem, how many retries you should have, and when you end and many others that you can see there. Now when it comes to instantiating the DAG, you start off by giving it a name. You'd also add a description, you'd pass in your default arguments and then your schedule interval. In this case, I'm making it here once a day and I wanted to start off two days back. It'll try to catch up by default, which means that as soon as I turn it on in this case, it would run a couple of times. Now, when it comes to expressing time, here are a number of presets that you can use as well as you can also use the cron type of notation. Now, when it comes to the tasks themselves, you can see here that t1, t2, and t3 are tasks and they are Python dictionaries. And note that we are instantiating for each of those the bash operator and then the Python operator.

In each of these, in the first two, we're running a bash command, you can see there that the first one is a date, then on the second one we are sleeping, and then on the third one we're simply calling a function with Python callable and that function is there at the top of this code block where it says sample function, and then in this case, we would simply return a 5. Now, you can

see this if you run it on the workflow platform and you look at the logs, you'd be able to see that output. The only remaining thing to do from that list that we took a look at of the number of things that you need to define is to set your dependencies, the order in which to execute the tasks. And you can see here that you could add a t2 to depend on t1 and you would specify it as shown. And, you could use the a different notation for the same where you have perhaps a more transparent way where you have t1 followed by t2 and between them you have your greater than less than type signs, in this case. Now, you could also specify brackets to be able to define tasks that will run in parallel. In this case, you can see here that t2 and t3 would run in parallel after t1 has completed. So, that defines the makeup of the number or the required structure for creating a DAG, and ultimately, as mentioned, it is simply a Python file.

Video 9: Creating a "Hello World" Workflow (07:44)

Let's write a simple "Hello World" workflow. We'll start off by creating a Python file. I'm going to go ahead and create a new file, then choose Python as its type. We'll then go ahead and save it for now into the sample directory, and we'll go ahead and call it abel.py. We'll go ahead and start off with the imports. So, you can see there we're importing Airflow and we are importing the DAG. Then we'll add our operators. Next, we'll do our simple function that we're going to be calling by one of the operators once we write our tasks.

Next, we're going to define our DAG, and you can see there that we're calling it abel. We're starting it two days ago, and our schedule interval is one. Next, we're going to go ahead and add our tasks. You can see there that the very first one is one that instantiates the bash operator, and all we're doing there is we're simply going to write out the date. You can see there as well that the handle to that is going to be t1, and that the task ID that we're giving it is called task one. After that, we're going to have another one that is going to be a bash operator as well, and we're simply going to be sleeping there for five seconds. And then the last one is going to be called task three, the task ID, and you can see there that that one is calling the function that we have here that simply returns five. Then the very last thing to do is that we're going to set the dependencies between the tasks, 1 will run before 2 and 3, and 2 and 3 will run in parallel.

So, let's go ahead and save that file. And next, we need to move that DAG that we just created and direct a cyclic graph into the DAGs directory. The reason we're doing that is because we're sharing that folder, that volume with Airflow, and Airflow is scanning that folder for new DAGs.

It does so every 60 seconds, so every minute. So, if you push years into it and you don't see it immediately, don't worry, just give it about a minute or so, and then it should show up. So, I'm going to go ahead and do that now. And as you can see there, I have moved it into the DAGs directory. As you can see here, I have logged in to Airflow, and I am on the landing screen, and you can see there that there is my DAG. Now, at times this can take longer than a minute to show up. But if you haven't seen it in say 10 minutes, you might want to restart your installation. And you can do so by going to your Docker dashboard, and in a minute that will show up here, going to the top of it and simply starting it, and starting it again.

Now, this is sort of the really heavyweight type of cycle. But in general, once you give it enough time, it should show up on its own. So, in this case, let's go ahead and select table, and we'll look here at the different ways of looking at the tasks. You can see there that we are on the tree view right now. You can see there that once the DAG starts, we have task one and then task two, and three run in parallel. Now, we can look at it in the graph view as well, and perhaps this one is a more transparent one, and you can zoom there as needed. There's other things that you can do as well. Once you start running, you can look at the calendar view, how long your task duration ran and so on. And you can look at code as well. So, here is our code that we wrote that is now inside of Airflow, inside of that folder that we just took a look at the DAGs folder.

And so now let's go ahead and come back. Let's enable our DAG, give it a chance to run, and we can go ahead and fire it as well manually. I'm going to go ahead and do so. And as you can see there, we get the two days that we're catching up. We started it a couple days back, and then we have triggered it manually as well and so that would be the third run. If we go ahead and take a look at the graph view, we can go ahead, and take a look at the logs of each one of these. And we'll go ahead, and take a look here at the very first log. We're scrolling down and you can see there that the command that ran on the bash was date, and that the output was the following on this system, on this date.

Now, of course that is dependent on what type of system you have. This really could be any time. In fact, one of the problems with machines is that at times you'll start at one odd times, so the date is certainly dependent on that. Now, if we go back to task two, and we take a look at the output of that, we can see here the same and in this case, this simply slept for five seconds, and nothing else was done. And then the very last one, the one that was calling that Python function, if we take a look at it there, you'd see there, this is the wrong one. If we look at task

three and we scroll down, you can see there that the function called and all that was returned was that value of five.

So, as you can see here, that gives you a sense here of what the user interface is, how to write a DAG, how to write a workflow and get it into the platform. You can see there that this is simply one. If you started to have many more runs, these are the runs that just ran today, we would be able to get here a much better sense of what's taking place. And if we walked into the workflow itself, we'd be able to see a graphical view. If you ever look at the timeline that you see in GitHub repositories of the activity and the errors, perhaps the heat that you might get, you get something similar here and those little squares would be colored by the following things that you see here. So, failed, up for retry, up for reschedule, and many of the other concerns that we typically have.

Now, that we have that we could see here the GitHub type of view that I just mentioned, you can see as well how many the tasks, how long the tasks are taking. Let's see if I can make a little bit smaller view here. You could see the number of tries. In this case, we don't have any of that data, but you could see that once you start to run at scale, this could be pretty useful. This would certainly give you a sense of how your tasks are performing, how your workflows are doing. So, now it's your turn. Go ahead and replicate the code that I have done. Make sure you can get it into your platform. Make sure you can run it and verify by looking at the logs the results.

Video 10: Creating a Loan Review Workflow (07:18)

Let's try to do a workflow that has more components. Let's try to mockup something like a loan review type of workflow. With that in mind, let's go ahead and move to the editor. We'll start off by creating a new file. We'll 'Select the language' of Python. We'll go ahead and save it. In this case into the sample directory that we have been working on. In this case, I'll call it loan_review. We'll start out as usual by doing the imports. Then the operators. Notice, there that we have an additional one. This case, in addition to the BashOperator, the PythonOperator, we have a BranchPythonOperator that we will use to make a selection between a couple of options. We will see that subsequently here in the code.

Then we're going to add some packages for some functions that we're going to be using. We'll then do our dag instance there. As you can see, we'll call it loan review. It'll start a day ago and it'll run every one day. Then we're going to start off with our tasks. And as you can see here, we

have three loans that we will be comparing, and we will be calling a Python function that will predict whether this is a good loan or a bad loan. And so, we have loan A, loan B, and loan C. After that we're going to have our branch that is going to select from those three loans by calling a function that we will write as well afterwards called best loan. And depending on that, it'll write either good loan or bad loan, in this case, just good or bad. So, let's go to those functions, the very first one. And actually, before we do that, let me go ahead and show you here the sequence. We're going to run loan A, B, and C in parallel, since those are simply going to give us a score and we'll see that in a moment how that is done.

Then we're going to have that branch, a task that is going to choose the best loan, and then we're going to write out good or bad depending on what we get there. So, the next step is that we're going to write our best loan. And actually, we need to also look at our prediction. So, our prediction, as you can see there, since we're doing a mock here, is simply a function that returns a random integer between one and ten. And we're going to take anything greater than eight, as you can see in the function above to be a good loan. Now from the very first three tasks that run, so loan A, loan B, and loan C, these here, we're going to get a value. The value is going to be that prediction, the result of calling the prediction function, the function that will return an integer between 1 and 10.

Now to be able to get access to the output of each of those tasks, we're going to use this parameter called ti, and we're going to pull from it the result of those tasks IDs. So, loan A, loan B, loan C. Again, this is what we called loan those blocks, those tasks. So, loan A, loan B, loan C. And then, we're going to create an array from that right of results, the predictions. And then from it we're going to pick the one with the best value or the highest value. Now here in this conditional statement, we're simply going to check and see if that value is greater than eight, and if it is, we're going to return good, otherwise we're going to return bad. So, that completes the logic that we're capturing here. Again, it's a mock of being able to select amongst a number of values, in this case three, those three loans, and then we're going to branch here choosing the best one.

We just saw the function that performs that, which is the function called best_loan. And then we're simply going to select good or bad depending on what the output is. So, let's go ahead and save this. And now, we're going to drag and drop that into our dags folder. If we go into it, we can see that it is there now. And now we're going to go ahead and wait for the Airflow

platform to pick up that DAG. And as you can see here, I have waited for my platform to load the DAG. I'm going to go into loan review. I'm going to go ahead and activate it. Actually, before I do, let me go ahead and show you here. You can see, how loan A and the options that it has. And if we look at the graph view, which is more intuitive in this case, you can see that loan A, B, and C is running in parallel. And then we have that branching and then the output of good or bad.

So, let's go ahead and activate this, and let's go ahead and run it or trigger it by hand. As you can see there, we immediately have a result, and we have two. Now let's go ahead and take a look at the graph view. And take a look at the output of each of these. We can go ahead and look at the log. This return four. So, let's go ahead and look at B. This was 10, so this would be considered a good loan. Let's go ahead and look at C. And in this case, we have 6. So, that would also be a bad loan such as the first one. Now we're going to go ahead and take a look here at the selection. And you can see here that one of those choices was good, the one that we had a ton. And then, you can see here that we have a good option and log capturing that result echoing good, you can see there.

And if we look at the other one, you'll see here that it did not run because it didn't meet, it didn't select that branch. So, as you can see there, that's a pretty representative problem of the type of wiring you might want to do. You can see here, how that is a very easy way to hook onto the functionality of what this platform provides you. As mentioned before, you would have to build this anyway yourself as you grew a script that was homegrown onto something that was relied upon by many people or perhaps many departments. And so, having this in place to be able to compose those workflows is something that is pretty useful. So now, it's your turn go ahead and make sure that you can run this example explorer on your own as well a few variations and get familiar with this platform as it is something that can be quite useful.

Video 11: Recap of Airflow: A Workflow Management Platform (01:46)

Airflow is one of those platforms that is surprising. It has a pretty simple proposition, managing workflows, and it might seem that that's not that big of a contribution. After all, many groups all over the world frequently do this themselves. They write a bunch of scripts that eventually grow up, but as they grow up, and they become more dependent or more departments become dependent on them, you find out that there's a lot more that you need to add to them. Well,

Airflow tends to be a great fit for that, and it turns out that it's something that we need frequently. Now, the directed acyclic graph model of bundling tasks into turns out to be a pretty good abstraction. And so in general, it is something that we're doing anyway. It is something that will simplify our work, and it is something that we can use to standardize what we're doing.

So, in general, we see there the simplicity of Python, the Python file, we see that we can scale it to go across multiple machines, we can capture history, we have a pretty friendly graphical user interface, we have additional REST interfaces, we can plug into a lot of infrastructure that we would like to. There's already a set of plugins and adapters that are available. And because of it, it has had a pretty good amount of traction, and it is widely implemented. So, now it is your turn, go ahead, and push on these concepts. Go ahead and try it on some of your projects that you might be working on. It is surprisingly easy to pick up, and it has a great fit, as mentioned to many of the things that we already do.