Course Overview

Course Overview

[Autogenerated] Hi, everyone. My name is smart but Tre and welcome to my course building your 1st 8 year pipeline with issued a breaks with exponential growth in greater volumes, faster data processing needs and dynamically changing business requirements. Traditional idiot tools face the challenge by the party spark can help us overcome this. Managing the spark environment is no cakewalk. Wouldn't it be great to have a cloud service? Just table that this course walks you through Educator Breaks, which is sparked based Unified Analytics Platform running on Microsoft Short, and you'll see how to quickly build the extract transform and Lord strips off your data pipelines. Some of the major topics that will cover include understanding the architecture and components of Federated Bricks setting of the as your data bricks environment. Building an end to an eternal pipeline on the platform, various ways to orchestrate the pipeline and other features like data bricks. AP Eyes and Delta like to help you build automated and reliable by plants. By the end of the scores, you'll be comfortable to work on a shooter bricks platform and build production really detailed pipelines. Before beginning the course, I would recommend being familiar with basic soft Microsoft ashore. The beginning courses in our library can quickly get you up to speed. I hope you'll join me on this journey to learn building. It'll pipelines with the building. Your first, it'll pipeline using your data \_\_\_\_\_ scores here in full sight.

Getting Started with Azure Databricks

Module Overview

Hi everyone, my name is Mohit Batra, and welcome to this course on Building Your First ETL Pipeline Using Azure Databricks. Before you even begin the course, you might be thinking, what's the need of Azure Databricks? If you are into ETL development, you would have surely used tools like Informatica or SQL Server Integration Services. They are great tools. Then why Azure Databricks? Azure Databricks is one of the services in Azure that allows you to build Apache Spark-based applications. Hey, wait a second. Where does Apache Spark come into the picture now? Apache Spark is the underlying in-memory data processing engine for Azure Databricks and is one of the most sought-after technologies in the big data world. So in order to build the ETL pipeline and better understand Databricks, you must first have a good understanding of what Spark is. So in this module, you will also learn about the basics of Spark. Sounds good? All right. And what exactly is Databricks? Databricks is an independent product. It is a fast, easy, and collaborative analytics platform, which runs on the cloud and allows you to build Spark applications without worrying about the challenges that comes with Spark. So in this module, we are going to dive in and understand the architecture and features of Databricks and how easy it is to get started. Next, you will learn about the different components of Databricks, which are going to be used for building the solution. All good so far, but is Databricks just a hosted solution on Microsoft Azure? No. The teams of Azure and Databricks, they came together to make it a first party service on Azure, bring enterprise-grade security, and have high speed connectors for Azure services. By the end of this module, you will be all set to learn how to build your first ETL pipeline using Azure Databricks.

Course Outline

Before we jump into the course, let's take a look at what you're going to learn in this course. You will start by learning about Azure Databricks environment, and you will see why do we need it, its architecture, features, and components. Since it is based on Spark, you will learn about the Spark architecture, RDDs, and DataFrames. And then you will see what Azure brings to the table and how deployment happens in Azure. Next, you will see how to set up the Databricks environment by launching a workspace, setting up a cluster, and creating a notebook. And you will also see how to set up the security. Then we'll start the ETL development journey. First you will see how to extract data from multiple sources, like Azure Storage and Azure Data Lake Store, mount these storage accounts easily, and use it in your projects. Then you will see different options and how to handle the schema. Following this, you will learn about doing data cleansing and transformations, how to create a derived column, do joins, aggregates, and much more. We'll then build the dimensions, facts, and different KPIs. Since failures can occur because of bad data, you will also learn about handling corrupt files and records. And then you will see how to graphically visualize the data and build dashboards for monitoring and end user consumption. Once the data is transformed and ready, you will learn about different ways of loading the data into the destination. We'll then go ahead and orchestrate the ETL pipeline to build an end-to-end workflow, schedule it as a job in Databricks, or run it via Azure Data Factory. And finally, you will get to know about other features like Databricks API, Azure connectors, and an introduction to a great component, Delta Lake, all of which would aid in building better ETL pipelines in Azure Databricks.

The Case for Databricks

Let's understand the need for using Databricks. Data engineering is an important aspect in the data space. It focuses on building a centralized data warehouse where data from multiple sources is collected and stored. It also keeps a track of the historical data. The raw data that has been collected is then validated, cleaned, and transformed into meaningful information that business users and data scientists can consume. It also goes ahead and build KPIs, or key performance indicators, that businesses can consume to make informed decisions. The process of handling and processing this data typically involves significant ETL operations. ETL stands for extract, transform, and load. This means you extract the data from a source system, like customer data, apply business-specific transformations, like combining their first name and last name, and load the data into the target repository. For doing this, there are lots of tools in the market like Informatica and SQL Server Integration Services. While these tools have been around for a long time and have catered well, especially for the structured data, but the challenge starts with increase in data volumes, especially today when the data is growing at an exponential pace. And it's not just the structured data that's increasing. Organizations also want to extract and process semi-structured and unstructured data like CSV files and JSON files, log and telemetry data, or extract data from NoSQL databases like MongoDB, Cassandra, Azure Cosmos DB, and much more. And while batch processing has been the traditional use case of ETL, there is an increasing need to process the streaming data as well. To handle such huge volumes and diverse data sources, you have to scale your infrastructure. While you may add more resources to your existing servers, it cannot be scaled up beyond a limit. This is where traditional ETL tools face the challenge, as they are not built to scale out to hundreds of machines. And lastly, data pipelines have to quickly adapt to the rapidly evolving business needs while maintaining accuracy and keeping the cost low. If any of these challenges sounds familiar, you have either faced some of them or foresee them in your projects, then welcome to the world of big data. Now you might ask, is ETL still relevant in the big data world? The answer is yes, because your business users, they still don't want to deal with the messiness and complexity of the raw data, be it traditional data or big data. And the success of a data scientist, it relies on solid data engineering to produce reliable data that they can work with. And this is where Apache Spark comes in. It is open source, and it's very popular in the big data community. Apache Spark is an extremely fast and powerful analytics engine for large-scale data processing, be it structured, semi-structured, or unstructured data. It is an in-memory engine, which means that it does the data processing in memory, and it can run workloads up to 100 times faster than Hadoop. It has a highly scalable architecture that allows it to run on hundreds and thousands of machines together and process terabytes of data in parallel. And Spark can also be used in variety of use cases, whether you are processing batch data, streaming data, doing machine learning, or advanced analytics. So Apache Spark can help us solve the challenges where traditional ETL tools fail. Sounds great, right? While Spark has got great features, a lot of developers feel it's hard to work with. The biggest challenge is the infrastructure management. Though Spark can run on hundreds of machines, handling the physical hardware, recovering from failures, patching the machines, managing the disks, or scaling out to meet the growing demands, all this is an extremely complex and costly affair. It also needs to be installed and configured on all the machines, which adds to the complexity. And all these challenges also make it difficult to upgrade to a newer version of Spark in production. And since Spark is only an engine, it requires setting up an ecosystem of tools for activities like development, deployment, security, etc. Spark does not have a native user interface, but there are other IDEs that can be used for development. And in big team setup, it's difficult to collaborate on projects. That's why we need an intuitive and collaborative environment in which we can easily work with Spark without worrying about the infrastructure and upgrades. And this is where comes Databricks. It has been founded by the same set of engineers that started the Spark project. While Spark is just an engine, Databricks is a completely managed and an optimized platform for running Apache Spark. It provides a whole bunch of tools out of the box so you don't have to plug in the basic components for Spark to work. That also means you can quickly start building your Spark-based applications. It also provides an intuitive UI and an integrated workspace where you can write the code and do real-time collaboration with your colleagues. And finally, the best part. It allows you to set up and configure the infrastructure with just a few clicks and manages the rest on its own, be it scalability, failure recovery, upgrades, and much more. So the processing capabilities of Spark powers the Databricks platform, and Databricks runs on top of Microsoft Azure cloud platform. So Azure brings all the features provided by an enterprise-grade cloud to the mix. Together, it forms a natively integrated first-party service on Azure called Azure Databricks. That's amazing, right? Now this all sounds great, but if you are coming from an ETL development background, you might ask, how difficult is this ETL transition? So let's have a look at a very basic ETL pipeline. The first thing you will do is extract the data. With a single line of code, you can start extracting the data from the source, be it any format or any storage that's supported. You will then start applying the transformations. You can limit the number of columns by selecting only few. You can add a new derived column to the transform data. For example, if your source contains FirstName and LastName columns, you can create a derived column called FullName by joining both of them. If you want to rename some of the columns, you can do that, and then you can also group and aggregate your data. After applying all the transformations, you will load the data into a target repository, again, with just a line of code. So, throughout the course, you will see various ways of extracting, transforming, and loading the data, along with various great features supported by Azure Databricks.

Spark 101

Since Azure Databricks is built on top of Apache Spark, so to better understand it, you must first have a good understanding of what Spark is. In the next few minutes, we'll learn about some of the basics of Spark. Apache Spark is an extremely powerful, in-memory analytics engine for large-scale data processing, which is built on cluster computing technology. So to work with Spark, you need to set up a cluster. You can install the Spark engine on a single machine, also called as node, or you can install and run it together on multiple nodes. Together, they constitute a cluster. Let's also understand the architecture of Spark. At the heart of Spark, there is Spark Core, which is the actual execution engine and all the other functionalities built on top of it. The Spark Core library provides the in-memory processing capabilities that allows it to run at massive speeds, and all the other functionality like memory management, fault tolerance, task scheduling, monitoring, and much more is provided by Spark Core only. Other great feature of Spark is that it natively supports multiple languages, so you can write your code in Scala, Python, SQL, R, or Java. And on top of Spark Core, there is a set of libraries that support different use cases. First one is Spark SQL. This module allows for structured data processing. This means if you have data in tabular format and you want to interactively query that structured data, then you will be working with this library. Remember, using Spark SQL does not mean that you only work with SQL language. You can write code in any language supported by Spark. A typical ETL pipeline that we want to build will majorly be using this library. Then we have the Streaming library for processing streaming data, MLlib library for machine learning, and GraphX library for graph computation. You can combine these libraries seamlessly in the same application. For example, you can integrate batch and streaming data processing with machine learning. The last thing you should notice is the mention of RDD at the Spark Core. RDD, or Resilient Distributed Dataset, is the fundamental data structure of Spark that is stored in the memory of the cluster. So RDDs are the in-memory objects in Spark. Think of RDD as a collection of elements or data, which is distributed to multiple nodes and stored in their memory. So when you write code to process the data, the processing happens on RDDs. There are four important features of RDDs, and let's discuss it one by one. As I just mentioned, RDDs are the in-memory objects. Let's understand that with the help of an example. When your code, which is running in the cluster, reads a dataset from the source, that dataset comes into the memory of the cluster and is called an RDD. Second feature of RDD is that it is partitioned. Looking at the same example, your cluster can have multiple nodes. So when you read the data from an external source, the data is partitioned, and each partition is stored in the memory of a separate node. So, to summarize, a dataset that you read is split into partitions, these partitions are stored in the memory of multiple nodes, and because it is stored on multiple nodes, Spark can process this data in parallel. And that's what is called as distributed data processing. And now you can understand. Spark can process huge volumes of data in parallel, and that's only because RDDs are partitioned. But remember, all these partitions together constitute an RDD. The third important feature of RDD is that they are read-only, which means once created, it cannot be modified. So you might think how will I do the transformation if it's read-only? RDD can only be converted into another RDD, or it can return the final result. To understand this feature, let's first understand that there are two types of operations that can be performed on an RDD, a transformation operation and an action operation. In a transformation operation, an existing RDD gets converted into another RDD. For example, if the first RDD contains customers' first name and last name and you want to combine them into a full name, it creates a new RDD. So transformation operation is like a function that takes an RDD as input and produces one or more RDDs as output. And because with transformation operation a new RDD is being created, this means you are defining a chain of transformations on a dataset, and this chain is called as a lineage graph. So loading the dataset from a source, converting the sales amount from INR to USD, or merging the first and last names to full name are all examples of a transformation operation. Now comes the interesting part. Transformations are lazy operations. What does this mean? This means a transformation or a chain of transformations, which is called lineage graph, they are never executed until and unless the second type of operation, which is the action operation, is performed. In other words, a lineage graph has been created, but it's actually not executed. Only when you apply an action operation here, like loading the data into a destination, then all the transformations in the lineage graph are executed and final result is produced, which in this case is load the processed data into the destination. So action operation is responsible for returning the final result of RDD computations. It triggers the execution using the lineage graph after optimizing the transformations applied to the dataset. This helps in running a highly optimal execution plan. So if you want to load the data into destination, show the output on a screen, or display the count, all these are examples of an action operation. So coming back to the read-only feature of RDD, by now you know that because RDDs are read-only, you can only apply transformations and actions on an RDD. Transformation creates another RDD and are not executed until action is applied. And once action is applied, the results are generated. One last feature of RDD is that they are resilient. RDDs know exactly how they are constructed by looking at the lineage graph. In case any node in the cluster crashes, RDD is automatically recreated and processed. So RDDs are resilient because of their ability to track their creation. In the event of a failure or a node crash, they can be reconstructed easily. This helps in providing fault tolerance to Spark applications. So, to summarize, RDDs reside and are processed in memory. They are partitioned and stored on multiple nodes in the cluster and processed in parallel. Since they are read-only objects, they can either be transformed to another RDD or return the final result. And they are resilient since they can auto-recover in the case of a failure. Sounds great, right? Now that you know the basic Spark architecture and RDD, let me introduce another layer on top of it, the DataFrame API. So what's a DataFrame? A DataFrame is just like a table you have in a relational database. It has got columns and rows. So using an API that allows you to work with a table-like structure is much more simple and straightforward. But you may be wondering, what's the relation between an RDD and a DataFrame? DataFrame is a high-level API built on top of RDD. Wow! That means all the great features of RDD also applies to DataFrames. So DataFrames are also in-memory, partitioned, read-only, and resilient. But not just this. In a DataFrame, data is organized into named columns, and it imposes a tabular structure on the data. Because it imposes a structure, Spark can now go ahead and apply a lot of optimizations. This gives you much better performance. So if you want more control over your dataset, you use RDDs directly, but if you are looking for better performance and less development effort, you use DataFrames API. For most of our structured data processing needs, like ETL development, DataFrame is good enough. So, throughout the course, we'll be using the Spark SQL library, Scala language, and DataFrame API.

What Is Databricks?

Now that you have a good understanding of Spark, let's understand what is Databricks. Databricks is a fast, easy, and collaborative Apache Spark-based Unified Analytics Platform that has been optimized for the cloud. Let me repeat that. It's an Apache Spark-based Unified Analytics Platform that has been optimized for the cloud. It has been founded by the same set of engineers that started the Spark project. Because it is based on Apache Spark, the data is distributed and processed in memory of multiple nodes in a cluster. All the languages supported by Spark are also supported on Databricks, be it Scala, Python, SQL, R, or Java. And it has support for all the Spark use cases, batch processing, stream processing, machine learning, and advanced analytics. But along with all the Spark functionality, Databricks brings a host of features to the table. First, and I believe the most important one, is the infrastructure management. Since Spark is an engine, so to work with it, you need to set up a cluster, install Spark, handle the scalability, physical hardware failures, upgrades, and much more. But with Databricks, you can launch an optimized Spark environment with just a few clicks and auto-scale it on demand. With Databricks, you also get a workspace with different users in the data analytics team like data engineers, data scientists, and business analysts can work together. They can share the code and datasets, explore and visualize the data, post comments, and integrate with source control. After you are done exploring data and building your data pipelines, Databricks help you to easily execute them on demand or automate to run on a schedule. And Databricks comes with a built-in access control and enterprise-grade security so you can securely deploy your applications to production. Let's have a look at the architecture of Databricks. It is divided into three important layers, the Cloud Service, the Runtime, and the Workspace. Let's understand these layers and their components one by one. First, the cloud service. Databricks is available on two most famous cloud platforms, Microsoft Azure and Amazon Web Services. Later in the module, we'll discuss why Azure is the preferred provider for Databricks. As you may very well know, launching a virtual machine, or a VM, in the cloud is extremely easy, and because Databricks runs on the cloud, it can easily provision the VMs or nodes of a cluster after you select their configuration. Databricks also allow you to launch multiple clusters at a time. This means you can work with clusters having different configuration, and whenever you create a cluster, it comes preinstalled with Databricks Runtime. We'll talk about Runtime in just a minute. And one of the great features of Databricks is the native support of a distributed file system. File system is required to persist the data. So whenever you create a cluster in Databricks, it comes preinstalled with Databricks File System, or DBFS. Important point to note is that DBFS is just an abstraction layer, and it uses Azure Blob storage at the back end to persist the data. So if users are working with some files, they can store the files in DBFS. Those files will actually be persisted in Azure Storage. Using this, the files are also cached in the cluster. So even after the cluster is terminated, all the data is safe in Azure Storage. One Azure Storage account that you see here is mounted to DBFS by default, but you can also mount multiple other Storage accounts like Azure Storage and Azure Data Lake Store. Once mounted, you can access them without providing credentials every time. You will see that in detail in the next module. The second layer is the Databricks Runtime. Whenever you are creating a cluster, you select a Databricks Runtime version. Each Runtime version comes bundled with a specific version of Apache Spark, some additional set of optimizations over Spark. In Azure, Databricks runs on Ubuntu OS, so Runtime comes with system libraries of Ubuntu. All the languages with their corresponding libraries are preinstalled. If you are interested to do machine learning, it preinstalls machine learning libraries. And if you provision GPU-enabled clusters, GPU libraries are installed. Good thing is that versions of these libraries that are installed with Runtime works well with each other, preventing the trouble of manual configuration and compatibility issues. Awesome, right? Before we go forward, two things you should note that we discussed earlier. In Databricks, you can create multiple clusters, and each cluster runs on a specific Spark version. This means you can run the same code on different versions of Spark, making it easier to upgrade or test the performance. The next part of Databricks Runtime is Databricks I/O or DBIO. DBIO is the module that brings additional optimizations on top of Spark related to caching, disk read/write, file decoding, etc. You can control these optimizations, but that's outside the scope of this course. But important point is that because of this, workloads running on Databricks can perform 10 times faster than vanilla Spark deployments. Now even though you can create multiple clusters in Databricks, doing so adds to cost, so you would want to maximize the usage of the clusters. This is where comes Databricks Serverless. Databricks Serverless clusters, or also called as high concurrency clusters, has got an automatically managed shared pool of resources that enables multiple users and workloads to use it simultaneously. But you might think, what if a large workload like ETL consumes lot of resources and block the short and interactive queries by other users? Your question is very valid. That's why each user in serverless cluster gets a fair share of resources, complete isolation, and security from other processes without doing any manual configuration or tuning. This improves cluster utilization and provides another 10x performance improvement over native Spark deployments. To use Databricks Serverless, you will have to create a high concurrency cluster instead of a standard one, which you will see in the next module. Databricks also provide native support for various machine learning frameworks via Databricks Runtime ML. It is built on top of Databricks Runtime, so whenever you want to enable machine learning, you need to select Databricks Runtime ML while creating the cluster. The cluster then comes preinstalled with libraries like TensorFlow, PyTorch, Keras, GraphFrames, and more. And it also supports third-party libraries that you can install on the cluster, like scikit-learn, XGBoost, DataRobot, etc. And a very interesting component here is Delta Lake. It was built by Databricks team and was called Databricks Delta, but this component is now open source and is called as Delta Lake. Even though many teams move to Data Lakes, but they struggle to manage them as the files in our Data Lake are still files, and they don't have the great features of relational tables. This is where Delta Lake comes in. Delta Lake is an open source storage layer. It brings features to Data Lake, which are very close to relational databases and tables, and much beyond that, like ACID transaction support where multiple users can work with same files and get ACID guarantees. Schema enforcement for the files. You can perform full DML operations like insert, update, delete, and merge. And using time travel, you can keep snapshots of data enabling audits and rollbacks. And there is much more. We'll understand this in more detail in the last module. The third layer in the Databricks architecture is the Workspace. It includes two parts. The first one is an interactive workspace. In this environment, you can explore and analyze the data interactively just like you open an Excel file, apply the formula, and see the results immediately. In the same way, you can do complex calculations and interactively see the results in the workspace. You can also render and visualize the data in the form of charts. In Databricks Workspace, you get a collaborative environment. Multiple people can write code in the same notebook, track the changes to the code, and push them to source control when done. And datasets that you have processed can be put together on a dashboard. It could be for the end users, or these dashboards can also be used to monitor the system. You will learn about these components that enable these features in just a minute. After you are done exploring the data, you can now build end-to-end workflows by orchestrating the notebooks. These workflows can then be deployed as Spark jobs and can be scheduled using the job scheduler. And of course, you can monitor these jobs, check the logs, and set up alerts. So in the same workspace, you cannot just interactively explore the data, you can also take it to production with minimal effort. So, to summarize, Databricks securely run an optimized version of Spark on cloud platform. You can create multiple clusters, and the cluster resources can be efficiently shared by multiple users and workloads. It brings together data engineering and data science workloads so you can quickly get started to build your ETL pipelines, handle streaming data, do machine learning, and much more. And it has an interactive environment for building solutions, sharing it with colleagues, and taking it to production, taking the game of data processing to a whole new level. Sounds exciting, right?

Databricks Components

Before we jump in and start building our ETL pipeline, let's have a look at the components you will be working with in order to build it. The first one, of course, is the cluster. In a Spark cluster, there are two types of nodes, worker nodes, the nodes that actually perform the data processing task. Since data in Spark is processed in parallel, having more worker nodes may help in faster processing. And driver node, which is responsible for taking the request, distributing the task to worker nodes, and coordinating the execution. There are two types of clusters you can create in Databricks, an interactive cluster that allows multiple users to interactively explore and analyze the data, and a job cluster that is used to run fast and automated jobs. In this case, a new cluster is created automatically, the job runs on top of it, and then it is terminated. To put it all together, when you want to set up a cluster, you will select the Databricks Runtime version, which is a VM image that comes with preinstalled libraries which we discussed earlier. You will select the configuration of a driver node and worker node separately, along with a number of worker nodes you need for processing. Databricks clusters also allows you to setup auto-scaling. So whenever the load increases on your cluster, it adds a new worker node and remove it when the load decreases. You can specify the minimum and maximum nodes, and scaling happen between these limits. And you can also specify when to terminate the cluster. That's a great feature. How many times it could happen that a user starts a cluster and forgets to terminate that? So by specifying the number of minutes, Databricks can auto-terminate the cluster if there is no activity. This can be a huge cost saver. But remember, auto-termination only applies to interactive cluster. Why? Because job cluster is anyway terminated as soon as the job ends. Next component is the Workspace. Workspace is a place where you will organize all the assets in a folder structure, be it notebooks, libraries, dashboards, or machine learning experiments. You can define fine-grained access control on all these objects, allowing users to use the same workspace, but only giving them restricted access. And there are multiple version control options available. For example, GitHub, Bitbucket, and Azure DevOps. So you have heard me a few times mentioning about notebooks. Notebook is the place where you will actually write your code, and you can do that in any Spark supported language. So you can write code in Scala, Python, SQL, R, or Java. And the great part about these notebooks is that you can write code in multiple languages in one single notebook. So you can extract data using Scala and write the transformation logic in SQL in the very same notebook. We'll see in the upcoming modules how that works. Also, one notebook can invoke the other one and pass the data. This can help in building end-to-end workflows. Using an interactive cluster, you can run queries in the notebook, or you can run the complete notebooks using jobs. These notebooks also support built-in visualization. So if you have data in tabular format, you can instantly visualize that using charts and graphs, or you can use the same dataset to quickly build a dashboard. And finally, as we have talked about it earlier, it supports collaboration. Then comes the jobs. Jobs allow the execution of a notebook, or if you have an external JAR file that you would like to execute on a Spark cluster, you can do that using jobs. A job can run immediately, or it can be scheduled. And by now you know that jobs can run on job clusters. Job clusters are created and terminated with the job, but if you have a running interactive cluster, you can run these jobs on them as well. Each job can also have a different cluster configuration on which it can run. This allows to use a smaller cluster for the smaller jobs and a large cluster for the bigger ones. And finally, you can fully monitor these job runs, retry on failures, and set up alerts for notification. Oftentimes, you would want to use third-party libraries in your projects. You can install these libraries on the cluster, and they can be in any Spark-supported language. Once installed on the cluster, you can refer to these libraries into your notebooks. A library can be scoped at the cluster level, which means it only exists in the context of a cluster, or you can install and scope the library at the notebook level. If you are coming from a relational database background, you'll be really excited to see this, that you can create databases and tables inside these databases. But be cautious. It is very different from a relational database. A table in Databricks represent a collection of structured data. This means the table has a structure, it has columns, and columns have datatype. This table is equivalent to a DataFrame because a DataFrame also has a structure. This means any operation that you can perform on DataFrame, you can do the same on a table. A table is created using the file present on the storage. So, in effect, it's just a representation of an underlying file where you know the schema. Any change in the file will also affect the table. Once the table is ready, you can query that or write to it. This is equivalent to working with a file. But you might think, if it's only representing files, then what's the use of databases and tables? The benefit is that you can keep multiple files together where schema is defined. This makes working with these files much easier. Also, you will see in later modules how you can use tables to pass around the data.

What Is Azure Databricks?

All right, now that you know a lot about Databricks, let's look at the final piece of the puzzle. What is it that Azure brings to the table? Databricks is not a marketplace app on Azure. The teams of Azure and Databricks came together to make it a managed first-party service on Azure. Databricks is natively integrated with Azure and its services. This also means Azure SLA applies to Azure Databricks as well, which is 99.95 % of time. And you also get the technical support for it, depending on your support plan. This is a big deal for organizations because Databricks service is fully backed by Microsoft. Next, Azure transparently deploys the Databricks workspace, clusters, and most of the resources in your own subscription, even though those resources are locked and you can't modify them, but you can track those resources in terms of usage and billing. Being a native service, Azure Databricks gets enterprise-grade security. It is fully integrated with Azure Active Directory and provides role-based access control, so you don't have to manage the users and their access separately. Super awesome for administrators. And finally, you get unified billing. You pay for usage of Databricks, for storage, and for VMs and disks created as part of the cluster all through a single bill. This may matter less to a developer, but for organizations, it's super important. Let us now understand how Databricks resources are deployed in Azure. There are two high-level components, the control plane and the data plane. The control plane resides in a Microsoft-managed subscription, while the data plane is in your own subscription. Whenever you create an Azure Databricks workspace, a Microsoft-managed Virtual Network, or VNet, is deployed in the control plane along with Databrick services like Databricks UI, job service, cluster manager, and notebooks. On the other hand, another Microsoft-managed VNet is also deployed in the data plane. A network security group is attached to handle the inbound and outbound traffic, and an Azure Blob storage account is provisioned that is used for Databricks File System, or DBFS. The control plane VNet and the data plane VNet are securely connected to each other. Now, when you want to work with Databricks, you will have to sign in using Azure Active Directory. Based on the permissions, you will get access to the workspace. Now, when you want to set up a cluster, the cluster VMs and the disks will be deployed in the data plane's VNet. This means the data is processed and stored in your own subscription. The important point to note here is even though data plane resources are in your own subscription, they are completely locked, and you can't make any changes to them. This is similar to how other Azure first-party services operate. The goal is to provide transparency by deploying it in your subscription, but making it easy to use and avoid any unintended changes to these resources. And the cherry on the cake, Azure has several high-speed connectors to its services that you can use with Databricks, like Azure SQL Database, Data Lake Store, Blob storage, Cosmos DB, Event Hubs, SQL Data Warehouse, Power BI, and much more. You will see the usage of some of them in this course.

Summary

Wow! We covered a lot in this module. We looked at how the growth in data volumes and data sources are becoming a challenge and how Apache Spark can help address them. We went into some of the basics of Spark, learned about the concept of RDDs and DataFrames, looked at the use cases and supported libraries. In this course, we'll be focusing on the Spark SQL library. We then saw how Databricks can take Spark to the masses with its unified platform and provides a bunch of features that enable collaboration between data analytics team members, as well as faster development using workspaces and notebooks. It frees us from managing the infrastructure with built-in support for our creating, terminating, and scaling multiple clusters while handling physical hardware, recovering from failures, managing security, and much more. It also allows quick production deployments via jobs, run them on schedule, and monitor them. And in between all this, it brings its own set of optimizations via DBIO, high concurrency clusters, etc. that allows it to run several times faster than traditional Spark deployments. And the Databricks Runtime is a host for several preinstalled components like Spark, machine learning libraries, and GBU libraries, as well as built-in frameworks like Delta Lake that helps in quickly setting up the environment and build better solutions. And finally, Databricks runs on Microsoft Azure platform as a native service and gets full features expected by a first-party service. It allows for transparent deployments, full integration with Azure Active Directory, native high-speed connectors to other Azure services, SLA guarantees, technical support, and unified billing. This makes Azure the go-to platform for running Databricks. Now I'm pretty sure you must be itching to see Azure Databricks in action, so see you in the next module where we'll start the journey to build our first ETL pipeline using Azure Databricks.

Setting up Your Databricks Environment

Module Overview

Hi, and welcome to this module on Setting up Your Databricks Environment. Now that we have a solid understanding of Azure Databricks, it's time to get our hands dirty. We'll start by setting up an Azure Databricks workspace. We'll then see the different types of clusters and how to create them. You'll also see what's a cluster pool and how useful it is. We'll then go ahead and create a notebook, attach it to a cluster, and run it. You'll also see how to use it, followed by some really cool features. We'll then go into the security aspect, how we can configure that, and what are the different options available. And finally, we'll walk through a scenario that we'll be using throughout the course. So let's get going.

Setting up Workspace

We saw in the previous module what's a workspace. Let's start here by understanding a bit more about it. We'll then go ahead and set up the workspace. Workspace is a fundamental unit of isolation in Databricks. Each workspace and its resources are completely isolated from other workspaces, even if they are in the same subscription. When you create an instance of Azure Databricks service, one workspace is created. The created workspace is identified by a unique workspace ID. As part of Workspace, several resources are deployed in control plane and data plane. As you saw in the previous module, data plane components are in your own subscription. These resources are locked and cannot be modified. Once you launch the workspace, you can organize various assets and folders like notebooks, libraries, and dashboards, as well as clusters and data, and you can even define fine-grained access control on all these assets. Let's see how to create a workspace through Azure portal. You can sign into Azure portal by going to portal.azure .com. Here I'm going to search for Databricks service and create a new instance. Add a name for the workspace. Let's keep it as PluralsightWorkspace. Select the subscription. Since all Azure resources reside in a resource group, so let's create a resource group with the name PluralsightDemoRG. And select location as East US 2. There are two pricing tiers that we can select from, Standard and Premium. Premium tier includes all features of Standard tier, as well as role-based access control. So let's select the Premium tier and create the workspace. The workspace is now ready. Before launching the workspace, let's see an interesting thing here, and that is the Managed Resource Group. This resource group has been created along with the workspace in our subscription and is locked for any changes. If you navigate to this resource group, you'll see three resources here. These are the data plane resources. It contains a virtual network deployed in the data plane, a network security group for managing the inbound and outbound traffic, and a storage account, which is the underlying storage for DBFS. Let us now launch the workspace. Carefully notice that it is using Azure Active Directory single sign-on to log in to Databricks platform. Once logged in, notice the URL. It's in the format of deployment region .azuredatabricks .net, followed by an identifier. This identifier is the workspace I'd, so next time you can directly log in to Databricks via this URL. What you see now is the workspace UI. On the left-hand side, you see the option of organizing all the assets in the Workspace tab, manage the databases and tables through the Data tab, create and manage the clusters via Clusters tab, and deploy the jobs in the Jobs tab. Go to the Workspace tab, and here you can see the options for creating a notebook, library, folder, or a MLflow experiment. You can import some code files, and you can even export all the files in the workspace. Let's create a folder, PluralsightDemo, and we can create subfolders or any other asset in this folder.

Creating Cluster

The second step in setting up the environment is to create a cluster. There are two types of clusters. An interactive cluster, which is majorly used to interactively analyze the data using notebooks. These clusters are created by users directly or by calling cluster API. But remember, they do not automatically terminate. You'll be charged while they are running, even if you are not using them. But Databricks allow you to auto-terminate these clusters if they are inactive for a certain period of time. This can provide huge cost savings. Because they keep running, any queries submitted are quickly executed, and you can auto-scale out and scale in based on demand. These are costly as compared to the second type of cluster, the job cluster. A job cluster is used to run automated jobs. This means you specify a cluster configuration while setting up a job. As soon as the job starts, the cluster is created and terminates when the job ends. Since they terminate with the job, the auto-terminate option is not applicable here. There is an overhead involved for job to start a cluster, but it provides high throughput because all resources are dedicated for the job. It can auto-scale on demand and are much cheaper than interactive clusters. Further, interactive cluster can have two modes. Standard mode clusters are meant for single users. It does not provide any fault isolation. This means if multiple users are working on a standard cluster, failure in code execution of one user may affect other users as well. It also does not provide any task preemption, so one running workflow may consume all the resources, thereby blocking queries from others. That's why it is recommended that each user work on a separate cluster. Lastly, standard mode supports all languages. On the other hand, Databricks Serverless, or high concurrency clusters, support multiple users. They provide fault isolation by running each user's code in a separate process. Even if some users are running heavy workloads, the others get a fair share of resources that allow their jobs to complete on time. This helps in maximum utilization of the cluster, thus helping to save cost. On the downside, it only supports Python, SQL, and R, but does not support Scala. Now let's see how we can create a cluster. To start creating a cluster, go to Clusters tab. Here you see the list of interactive and job clusters, but as you know, you can only create an interactive one. Let's click on Create Cluster. Provide a name here. Let's keep it as DemoCluster. You can select the cluster mode, Standard or High Concurrency, and by now you very well know the difference between the two. I'm going to select the Standard mode since we'll be working with Scala. We'll leave the Pool option for now, but we'll come back to it later. Next, you need to select the Databricks Runtime version. In the last module, we discussed in detail about it. Databricks Runtime is the VM image that comes with preinstalled libraries, which has a specific version of Spark, Scala, and other libraries. One thing you should note is the different configurations of a runtime. For building an ETL pipeline, you can select the version 5.5, which is the latest version at the time of recording this course. If you want to enable machine learning, you can support 5.5 ML. That will preinstall ML libraries. If you want to use GPU-accelerated VMs, you can select 5.5 or 5.5 ML with GPU, and it preinstalls GPU libraries. You can now go ahead and select the configuration of a single worker node. These are different VM sizes provided by Azure. Depending on your requirements of memory, cores, and hard disk, you can select the configuration. Remember, all the runtime libraries will be installed on each worker node, and then you can select number of worker nodes you need for your cluster. Let me select three here. After selecting the worker node configuration, you can now select the configuration of the driver node. You may have also noticed DBUs mentioned with each configuration. So what's a DBU? DBU stands for Databricks Units and is a unit of processing capability per hour. Each configuration tells you how much DBUs will be consumed if VM runs for 1 hour and you pay for each DBU consumed. To understand how much worker nodes you need, you will need to run your workload, and by trial, you can do the capacity planning. Or you can enable autoscaling, provide the minimum and maximum number of worker nodes, and let Databricks handle that. This can help you quickly figure out your requirements. And finally, you can enable auto-termination of cluster by providing the number of minutes. Let's select 30 minutes, and if there is no activity for 30 minutes, the cluster will auto-terminate. Hit the Create button to finish creating your cluster. Azure will now go ahead and provision the required VMs with specified configuration and libraries as specified by Databricks Runtime. Once the cluster is up and ready, you can terminate, restart, or delete the cluster at any time. You can even edit the cluster by selecting it, clicking Edit, and changing the cluster configuration. Remember, changing the cluster configuration may require a restart of the cluster. For a particular cluster, last two things which you should note for now is the Event Log, and the Driver Logs. Event Log shows you all the events that have happened with the cluster. For example, when the cluster was created, where it was terminated, if it's edited, or if it's running fine. This helps to track the activity on a cluster. And in the Driver Logs, you will get the logs generated within the cluster, notebooks, and libraries. Now there is one thing we left, and that is the pool, which you saw while creating the cluster. Even though you can save cost by terminating a cluster when not in use, booting up a cluster might take few minutes, and this can be annoying sometimes. So you can create a pool of idle, ready-to-use instances, and you can attach multiple clusters to a pool while creating them. So if cluster one needs two instances, pool allocates the idle instances to cluster one. If second cluster needs two instances, pool creates a new instance and then allocates two instances to the cluster. After the cluster terminates, the instances are returned to the pool where they can be allocated further. If cluster one scales and needs one more instance, it can be allocated from the idle instances in the pool. And that's why pool helps in reducing the cluster start and autoscaling times. Now there are four important properties of a pool. Idle instance auto termination. This means you can define if you want to terminate an idle instance after a certain number of minutes. Minimum idle instances. Using this, a certain minimum number of instances will always be running as part of the pool, regardless of auto-termination minutes specified. Maximum capacity. This property can be used to define an upper limit of instances in the pool. If clusters demand more than this capacity, it will result in a failure. And finally, you can define instance type. If a cluster is attached to pool, the worker and driver nodes has to use the same instance configuration as specified in the pool. Let's see how you can create a pool and attach it to a cluster. In the Clusters tab, go to Pools, and create a new one. Provide the values for properties you just saw, name, minimum idle instances, max capacity, minutes for idle instance auto termination, and the instance type, and create the pool. And you can notice the used and idle instances. Let's create a new cluster. Let's name it as PoolBasedCluster and select the pool, DemoPool here. Notice that you no longer have the option to select the worker and driver Node configuration. It's the same as the instance type of pool. Fill up the rest of the properties and create the cluster. So whenever this cluster starts or auto-scales, it will pick up driver and worker instances from the pool, reducing the cluster start and scale time. Very useful indeed, right?

Working with Notebook

Now let's take a quick tour of the notebook and see how to work with it. Start by creating a notebook under any folder, provide a name, and you can select any language, Python, Scala, SQL, or R, in which you want to do development, and I'm going to select Scala here. The Cluster drop-down will show the list of all running clusters, and you can decide on which cluster you want to execute your code. I'm going to select DemoCluster here, which we created earlier. You can see that the notebook is attached to the cluster. You can detach the notebook from the cluster and attach it to any other cluster. This allows you to run your code on different versions of Spark. Remember, it's not mandatory to have a running cluster to do development. You can write code in the notebook and later attach it to the cluster to execute it. Now that you are all set, you can start writing the code in the cell. Let's check here if everything is working fine by writing the first string, Hello World. To execute this, you can use the Run drop-down and select Run Cell. It went ahead and executed the cell, created a String variable implicitly, and assigned it the Hello World value. You can also insert a new cell. Let's define a sum variable explicitly, and you can also execute this using keyboard shortcut Shift + Run. So a notebook is basically a collection of runnable cells. You can execute a single cell or the whole notebook at once. The notebook also preserves the state, so you can use the same variable at any other place in the notebook. Now let's look at some of the cool features of notebooks and get used to them. First one is auto-complete. You can start writing and press Tab. This will bring the drop-down from which you can select commands, variables, et cetera. Interesting, right? Next, in the Revision history, you can see all the changes that have been made to the code, and you can select the previous version and restore your notebook to that version. The other important thing that you need to do is to enable version control. You can either download the notebook and manually add it to the source control, or you can enable source control right inside your Azure Databricks workspace. Navigate to Account, User Settings, select Git Integration, and select either GitHub, Bitbucket, or Azure DevOps. Once enabled, you can link your notebook to your Git repo. Very useful, right? And you have heard it many times, notebooks enable collaboration. Multiple people can write code in the same notebook at the same time, and you will see this icon if someone else is also working on this notebook. And you can select code, use this icon, and post comments. This is extremely helpful in a shared environment. Let's see one more thing. Let's write a command in a new cell, %md ### Exploration Notebook, and let's mention testing new commands. Come out of the cell, and magic. Now you have a cell where you can write detailed description. To edit the comment, double-click on the cell. Any command starting with percentage sign are called magic commands in Databricks. %md is used for documentation, and you can even include text images or links. Extremely helpful to document the notebook. You'll see more magic commands as we go forward.

Configuring Security

Before you get down to writing any code, it's important that you understand and configure the security controls. I would categorize the security into three layers. First one is of the infrastructure that you are deploying. This has been taken care by Azure. We saw earlier how workspace is secured by deploying the sources in control plane, and data plane virtual networks, which are securely connected. Traffic is managed by security groups, and resources are locked to prevent any changes. Second one is the identity control. Users are authenticated using Azure Active Directory single sign-on, so to log in, user must be a part of Azure Active Directory and must be added as a user in Databricks Admin Console. You'll see that in a demo in just a minute. Third one is the fine-grained user permissions on Databricks assets like clusters, folders, notebooks, jobs, and data. Let's see how we can set it up. In Databricks Workspace, go to Account, Admin Console. From here, you can add users who can access the workspace. Let's first add a user, user1@ pluralsight.com. Because this user does not belong to Azure Active Directory, this user will not be able to log into the workspace. Let's add another one which is part of our Active Directory, demo@ pluralsight.com. This user is successfully added and can now log in to the workspace. Let me add a few more quickly. After you are done adding users, you can define which users can have full permissions by making them admin and who are allowed to create a cluster by setting this permission. And you can even organize users in groups and give access at the group level. Let's now set up the permissions at the folder level. In the workspace, open the drop-down of a folder. Go to Permissions, select a user, and the permissions you want to assign to that user. These permissions are inherited down to the subfolders and all the notebooks present inside them. And you can even set up permissions at individual notebook level. Select a notebook, and you can assign the permissions in the same way as folders. With read permission, users can view the notebook and make comments. Run permission allows you to attach or detach a cluster to the notebook and run the commands. With edit permission, you can make changes to the notebook. And, of course, manage permission allows you to change the permissions at the notebook level. To give fine-grained permissions at the cluster level, go to Clusters tab and select the Edit Permission option for a particular cluster. There are three types of permissions, Attach To, Restart, and Manage, which are self-explanatory. Manage allows complete control over the cluster. Setting the permissions on jobs are similar, but you will see that when we start creating jobs. So you have seen that Databricks comes with extremely simple yet powerful security controls that allow you to manage fine-grained access control while it itself manages the infrastructure security and user authentication.

Scenario Walkthrough

Now that our environment is ready, let's walk through the scenario that we'll be addressing. Globomantics is an organization responsible for processing New York City taxi service data. There are two types of taxis in New York, yellow taxis and green taxis, and they collect ride-related information, like pickup and drop time, details of pickup and drop location, trip distance, passenger count, as reported by the driver, if it's a solo or a shared trip, as well as payment-related information, like fare amount, extra charges, tips, toll tax, total amount, et cetera, and the mode of payment, be it cash or credit card. They receive this data once a month in the form of CSV files. In their on-prem setup, they extract this data using an ETL tool, make transformations, and build dimensions and facts. Then they store this data in a data warehouse. Following that, they build monthly aggregated reports and KPIs like revenue collection by green and yellow taxis, or by their pickup location, or total trips being taken, or maximum trips per region, and much more. Though the system was working well, the challenge started when Globomantics started collecting the data of FHVs, or For-Hire Vehicles. FHVs are of multiple types, community cars, black cars, and luxury limousines, and also high volume for-hire services, which dispatches more than 10, 000 trips per day and include app-based companies like Uber and Lyft. The request for FHVs by passengers are accepted by bases, and then the bases dispatch the drivers. There are more than 750 bases and 100, 000 FHVs, and all these different types of FHVs operate in different ways. The number of FHVs are much higher than yellow and green taxis, which led to exponential increase in data volumes. The schema of FHV data is different, and data started coming in not just CSV format, but TSV and JSON format as well. The requirement is to merge the data of all types of taxis, compare them against one another, and build granular and aggregated reports. And waiting for a month is no longer an option. The requirement is to process the data at higher frequency. And then raw data also must be preserved for enabling data science scenarios. That's why Globomantics has decided to build a Data Lake to keep the raw, as well as processed data together. The raw data is to be captured in Data Lake. They have decided to use Azure Databricks to profile, merge, and process data of all types of taxis and then build common dimensions, facts, and KPIs, as well as reports. Once built, these entities need to be stored back to Data Lake, and this would also enable business analysts and data scientists to consume the data.

Summary

So in this module, we went through many components of Databricks and see how to set up them. This will act as a premise for the following modules. We saw how to create a workspace and organize all the assets like folders and notebooks. We then went ahead and created an interactive cluster and learned how it's different from a job cluster. It has two modes, standard and high concurrency, and we went through various configuration settings: Databricks Runtime, autoscaling, auto-termination, et cetera. And then we saw pool, which contains a set of instances ready to be used by the attached clusters, and that helps in speeding up the cluster start and scale times. And then we created and attached a notebook to the cluster and saw some cool features: autocomplete, revision history, Git integration, posting comments, and magic commands. And finally, we went into configuring the security for all these assets, and we also worked through the scenario of New York City taxi service that we'll be working with. So let's get down to building the ETL pipeline by extracting the data in the next module.

Extracting Data from Multiple Sources

Module Overview

Hi, and welcome to this module on Extracting Data from Multiple Sources. Extracting the data is the first step in our ETL pipeline. We'll start by understanding various data sources supported by as Azure Databricks. We'll see how to mount the Azure Storage and Azure Data Lake Store to DBFS. Since mounting an Azure Data Lake Store requires service principal authentication, you'll see how to set up and use it. We'll then go ahead and start reading the data. You'll see how to explore and extract data from different file formats, and you'll also get to know many options that you can use for extraction. And finally, we'll see how to infer the schema from the files or manually build it and apply them to DataFrames. We'll be looking into a lot of options, so let's get going.

Extracting from Azure Storage Services

Let's start by understanding how we can extract data from different Azure Storage services. But first, you might ask, what kind of data sources we can extract from? Azure Databricks allows extraction from any relational database that supports JDBC. It has specialized connectors for certain NoSQL databases like MongoDB, Neo4j, Cassandra, and more. And along with that, it has high-speed connectors for Azure data services, which are either built-in or freely available like Azure SQL database, SQL Data Warehouse, and Cosmos DB. but what we are most interested in is the Azure file storage services, Azure Storage and Data Lake Store. To use them, you can mount it to DBFS. What does that mean? As you know, DBFS is the file system of Databricks, so mounting a file-based storage to DBFS allows seamless access to data from the storage account without requiring credentials. Think of this as mounting a network drive to your computer. So you only need to provide credentials for the first time while mounting storage, and thereafter, you can access it without credentials. And instead of using URLs, now you can use file semantics as if they are local files. But even though you are interacting with DBFS, the files are actually persisted to storage, so they are safe even if you delete the cluster, but even the workspace. Azure Blob storage can be mounted by using an access key or restricted shared access signature, and Azure Data Lake can be mounted using service principal. If you remember from last module, one Azure Storage is mounted to DBFS by default. These are additional storage accounts we can mount to DBFS. So, in the demo, you'll see how to mount Blob storage and Data Lake Gen1. Let's start with Data Lake first, but for this, you need a service principal. So what's a service principal? Think of this like a service account. It's an identity which can be used by applications so that it can access Azure resources. So, instead of putting your own AD credentials in an application, you should use service principal to access as your resources. In order to use it, you need to create an Azure AD application and then create a secret key that acts like a password. Then you can give access to this Azure AD app on Azure resources and use app ID and secret to access them. Let's first create a service principal. In Azure portal, navigate to Azure Active Directory, App registrations to reach this page. Register a new application. Let's provide the name as PluralsightServicePrincipal and click Register. Once registered, open the application, and notice two important attributes here, the application ID and the directory ID. Copy these values because it will be required for mounting. Next, go to Certificates & secrets and generate a new client secret. Copy and save the generated value as you won't be able to retrieve it again. Now, that the service principal is ready, let's give it access to Data Lake Gen1 account. Switch over to Data Lake account. Before we go forward, copy the ADL URI, which will be used to access the account. In the account, go to Data explorer. At the root folder, select Access. From here, let's provide the access to the Azure AD app we just created. In Select user or group option, search for the Azure AD app, PluralsightServicePrincipal, and select it. Next, select the permissions. For the demo purpose, let's select all of them. This gives access to the app on Data Lake. Now that the access is granted, let's switch back to the Azure Databricks workspace. Let's create a setup notebook to keep all setup-related information, including the mounts, in one place. To mount Azure Data Lake, let's put all the information we have collected so far as the config. In client.id, add the application ID. In credential, add the secret value. Remember, we are putting the secret as plain text. This is not recommended, and you should either use Azure Key Vault or Databrick secrets to avoid this. And then add the directory ID in the URL. Now that the config is ready, let's use dbutils.fs .mount to mount the Data Lake. There are three things that you need to provide. First is the source. This is the ADL URI that we copied. Second, select the mount point. Let's put it as /mnt/datalake. Remember, to access the files in the Data Lake, you'll now be able to directly use this path without any credentials. And third, provide the configs. Let's execute the cell, and that's it. Data Lake account has now been successfully mounted to DBFS, but I'm pretty sure you must be having some questions in your mind. First, who can access that mount? The mount can now be accessed via any cluster or any user in the workspace. Okay, and what is this dbutils? DBUtils, or Databricks Utilities, are a set of utilities to perform powerful tasks inside notebooks. There are various types of utilities available, like for file system, for handling notebooks, for using libraries, and much more. As an example, the DBUtils can be used to mount a storage account that you just saw. It can be used to invoke one notebook from another one, or it can be used to install a library on the cluster from the notebook. There are a lot of things you can do using DBUtils, and you'll continue to see that during the course. Let's see how we can use these mounts in our exploration notebook. To see all the files and folders in the mounted Data Lake, let's run the command dbutils.fs .ls and provide the mount path, /mnt/datalake. And you can see path, name, and size of all files and folders without the need for passing any credentials. Let's see what's present in the source folder. We have one file here, TaxiZones.csv. To quickly see what data is there in this file, let's use a command, dbutils.fs .head, and it shows a part of the file. Super useful. But before we start extracting data from these files, let's mount as your storage as well. Let's switch back to the portal. As I shared earlier, Azure Storage can be mounted using access key. So navigate to Storage account, Access key, and copy key1 or key2. Note the name of the container in which the source data is present, which in this case is taxisource. Back to Databricks workspace, mounting Azure Storage is similar to Azure Data Lake Store. In the configs, add the access key you copied earlier, and add the name of the storage account. Then use dbutils.fs .mount command again, add the name of the storage account, and add the name of the container as well. Execute the cell, and this will mount Azure Storage now to DBFS. As simple as that. Amazing, right?

Reading Multiple File Formats

Now that we are done mounting the storage accounts, let's start reading the files. At first, we are going to read yellow taxi and green taxi data from Data Lake. For this, let's create a new notebook and name it as ExtractYellowAndGreenTaxiData. Earlier you saw a command, dbutils.fs .ls, to list the files in the mounted location. Let's check that again. Databricks also has a magic command, %fs for the file system. That acts as the shortcut to dbutils.fs library. This means you can use %fs ls followed by the path to display the list of files. Interesting, right? Okay, let's start by reading a file with yellow taxi data. You can explore the data by reading the initial part of the file by using a head command, and by now you know that you can also use dbutils.fs .head command to do the same. Let's execute the cell, and it looks like a CSV file with these columns. So to do further processing, extract this file as a DataFrame now. You can run spark .read .csv followed by the file path. Few things to note here. What's a spark? Spark is a variable using which you can access SparkSession, and SparkSession is the entry point to Spark SQL. All the Apache Spark-related information can be accessed from the SparkSession. Next, to read a dataset from the external source as DataFrame, you need to use this read property on SparkSession. And notice we are specifying a CSV method that will go ahead, load a CSV file, and return the result is a DataFrame. Let's execute this. And if you recall from last module, reading a data from an external source is a transformation operation on RDD or DataFrame. This means regardless of the size of this file, it will quickly create a DataFrame. Why? Because it has actually not read the whole file. That's why you can see how quick it was. It also figured out the structure of the file. So there are 17 columns in the DataFrame, but the names are not appropriate. But if I scroll up, you can see the first row in the file contains the column names. So let's use the option method to specify that it has a header. Execute the cell, and now you can see it has correctly picked up the names of the columns. So let's display the DataFrame contents. Here we are applying an action operation, which implies that result is being generated. So if you have a very large file, this command could take some time. Interestingly, you can even go ahead and download the contents of this file. And notice some important attributes here that we'll be using later. There is vendor ID, ID of the vendor that is collecting this data; pickup and drop time; passenger count, as reported by the driver; trip distance; rate code ID, which tells us if it's a solo trip, shared trip, or a trip to the airport; pickup and drop location ID, details of which we'll be extracting from another file; payment type, cash, credit card, et cetera; and the fare details. We'll keep discussing the schema and how we can use it in the later modules as well. Let's extract and display the green taxi data as well, which has similar structure. But the green taxi data has a TSV format. To extract this, only thing you need to do is change the delimiter. To do that, let's add an option of delimiter as tab, and executing this correctly reads a green taxi data. If you look at the trip data, it includes a payment\_type field. Let's extract details of this payment type from a file. First, let's check what's there in the file. Let's use head command and read PaymentTypes file. It looks like JSON data. So let's write the code to extract, and instead of using CSV method, let's use another one, JSON method. Notice that it has parsed the JSON and identify two fields, and data is now in a DataFrame. This is simple JSON data, but you will see how to process a more complex one in just a minute.

Applying Schemas

Because you are reading data from files in a DataFrame, you would want to apply a schema to it. Let's understand how to handle that with an example. Let's start by reading the FHV data in a new notebook, ExtractFHVData. FHV data is present in Azure Storage, and there are multiple files delivered in a single month. Let's read one file first. You can see that it correctly figured out the column names, but data types are all string. This is because by reading only few lines from top, it cannot tell what is the data type of the column. To identify the type, let's use another option, inferSchema and set the value as true. InferSchema reads the whole file to identify the data types. It took a bit of time, but you can see that it correctly identified the types. But it can quickly become a challenge if files are extremely huge. To solve this, you can create your own schema and attach it to a DataFrame. Import the required libraries, and define the schema. Let's name it as fhvTaxiTripSchema. Here we are creating a struct type object, and fields can be defined using StructField. So instead of inferring the schema now, you can attach a fhvTaxiTripSchema to the DataFrame. Once you execute it, see how quick it is. And this is only because it's no longer reading the file. This is very important when you deploy your pipelines. Another feature is that you can read multiple files at once. In the path, use the wild ard character, and all matching files will be picked up in a single DataFrame. Using this, you don't have to worry about reading a whole bunch of files separately. Isn't that interesting? All right, now notice that FHV data has a dispatch base number. Let's read these bases from another JSON file. Read the head to see what's there in the file. Okay, it's JSON data. Let's read that in a DataFrame. Execute it, and oh, there's an error here. And the reason for the error is it's a multiline JSON file. So, to read it, you can specify multiline option to true. It's now able to read the file and its schema correctly. But JSON files can have complex schema. So to make sure that all JSON files adhere to a specific schema, let's define one more. Now, we have created a nested structure. A field can itself be a structure. So an address field is of struct type and contains fields like building, street, city, state, and postal code. Apply the schema to the DataFrame. If any of these fields is not present in file, it will show up as null, or if any additional attributes are present in file, they will be ignored, but this will not throw up any error. Execute this, and see that Address and GeoLocation are complex types. You'll see how to read data in these types in the next module. So you have seen that you can either infer the schema or build and apply your own schema, and the schema itself could be a complex structure, and you can do that for one or more multiple files.

Summary

In this module, we covered a great number of extraction features and how Azure Databricks support relational, non-relational, and file-based storage types. We then saw how to mount Azure Storage using access keys and Azure Data Lake Store using service principal authentication. And you also saw the powerful set of Databricks Utilities, or DBUtils, that can be used to explore data in notebooks. We then read the data from multiple types of formats, CSV, TSV, and JSON. And we also use various options, be it delimiter option, header option, multiline option, and more. And finally, we saw how to infer the schema for the files or build your own schema and apply it to a DataFrame. And, of course, we can build simple or complex schemas. It's real fun to explore the data in such an interactive way. In the next module, you'll see how to apply various transformation functions on the data that we have extracted here and how to handle the corrupt data.

Transforming and Cleaning Data

Module Overview

Hi, and welcome to this module on Transforming and Cleaning Data. Now that we are done extracting data from multiple sources, it's time to profile, clean, and transform the data into required dimensions and facts and build reports on top of it. We'll start by understanding the dimensional model we want to build for our taxi service scenario. But before we start working with any files, we'll compare some of the common transformations available in other retail tools and what's available in Spark, both in Scala and Spark SQL. We'll then start using methods available in Spark and Databricks to analyze the source data and then use other set of methods to clean it by applying the data quality checks. Once our data is clean, we'll use Scala to apply business-specific transformations to build dimensions, facts, and reports. And we'll also see how to mix two languages, and that's where we'll be using Spark SQL. And at the end, we'll see various ways of finding the corrupt records in the source files. So let's start by understanding the dimensional model we'll be building. We have three fact tables, one for storing FHV taxi data, second for yellow taxi, and third for green taxi data. FHV taxi fact is linked to bases dimension, which tells us the dispatch base location of a taxi. All the three fact tables are linked to taxi zones dimension, which will tell us the pickup location and drop location details for each trip. Yellow and green taxi facts are linked to two more dimensions, rate codes dimension, which tells us if it's a shared trip, solo trip, or trip to any specific airport. On the other hand, FHV data only contains a flag to tell us if it's a shared trip or a solo trip. We can also go and link FHV data to records. The second dimension yellow and green taxi facts are linked to is the payment types, which tells us the mode of payment. We do not have any payment information for FHV data. Of course, there is much more that can be added. Remember it's a very minimal model we are building focused more towards the demo. Sounds good?

Understanding Common Transformations

Before we start applying transformations to our extracted data, you must be thinking, Does Spark have all the transformations supported by other ETL tools? So first, let's see some common transformation that you'll need. We'll see if they are available in Informatica and SSIS and how you can use it in Scala and Spark SQL. Sounds good? The first type of transformation is the filtering of data. In SSIS, you have Conditional Split transformation, and in Informatica there is Filter transformation. You can do the same in Spark by using where method on the dataframe and passing the filter condition, and it's pretty simple is SQL by using a where clause, right? Next, it's very common to create calculated columns. There is derived column in SSIS and expression transformation in Informatica. In Scala, you can use withColumn method, pass in the new column's name, and the calculation expression. And one of the ways in SQL is to create that directly in SELECT clause. Another one is counting of rows in a dataset. You can use Row Count in SSIS and Aggregator in Informatica and just the count function on the dataframe and SELECT COUNT \* in SQL. So far, you must be noticing that working in Scala is very close to working with SQL, and you are very much right. The next thing you would want to do is to aggregate the data. SSIS and Informatica support similar transformations, Aggregate and Aggregator. And in Scala, you can use groupy and provide columns to group on, and then use agg method to mention if you need sum, count, average, min, or max. And in SQL, you can use SELECT and GROUP BY to do the same. Next, to sort the dataset, there is Sort in SSIS and Sorter in Informatica. And to do the same in Scala, use orderBy method followed by the column to sort on. And to do the same things in SQL, use ORDER BY clause. You can specify if you want to sort ascending or descending in all of them. Now to join the two datasets and perform inner, outer, or full joins, use Merge Join in SSIS and Joiner in Informatica. And Scala syntax is very close to SQL. On first dataframe, use the join method and specify second dataframe to join on followed by the join condition and the type of join. And in SQL, you can join two tables by using JOIN clause followed by the join condition. One the other hand, if you want to combine two datasets, all of them have very similar transformation or method, Union. And of course, there are much more transformations. But the point here to notice, that you can use similar type of transformations in Spark as you use is SSIS, Informatica, or any other ETL tool, and even in Spark you can choose any language of your choice to build the ETL pipeline. Awesome, right?

Analyzing and Cleaning Data

Now that we have gone through the common transformations using an ETL process, it's time to apply them to our pipeline. The first step here is to analyze what's present in the data and clean it up so that business-specific transformations can be applied later. So let's see what data quality checks you can apply. You can start by checking if data is complete or not. This means any fields with must-have information should not be empty. For example, pickup and drop location for a taxi ride is important. If it's not present, you can remove the complete role. But in some cases, you can put up a placeholder to fill the missing values and analyze the missing data later. Then you can check for data uniqueness. This is very important and is a very common check. To make sure it is unique, you can remove the duplicate records and keep only one copy. You can then check for timeliness, which means if the data is within the expected date range or not. In our case, if pickup and drop time is in the past or future, you need to remove and process that data separately. And finally, you can check for data accuracy, which means if it logically makes sense or not. Let's see how we can apply these data quality checks. In our notebook, ExtractYellowAndGreenTaxiData, we have already extracted yellow taxi data for the month of December 2018 in a DataFrame. I have updated the command to infer the schema for yellow taxi data as well. After analysis, you should define the schema and apply it to the DataFrame. Spark has a great feature to analyze the numerical columns in the DataFrame using describe command. Let's apply it to some of the columns in the DataFrame and execute it. It analyzed the whole file, and you can see that it figured out five important attributes, count of records, mean value, standard deviation, and minimum and maximum value. This is super useful. At first, you can see that there are over 8 million records in the file. If you look at the passenger\_count, minimum value is 0. This, of course, tells us that data is inaccurate, so we can get rid of all the rules where passenger\_count is 0. Remember, we are doing this for demo purpose. In your projects, actual action will depend on your business requirements. And the maximum is 9. Since allowing more than five passengers are illegal, a KPI can be built to provide records with more than five passengers. Okay, and in the same way, minimum trip\_distance is 0. It needs to be more than 0 so you can filter these records as well. That's how you can analyze rest of the columns in the DataFrame as well. Let's clean up the records in the DataFrame based on what we have discussed. To do that, apply where clause on the DataFrame to only get the records where passenger\_count is greater than 0. Easy, right? Apply the condition for trip\_distance as well. Few things to note here. You can use filter clause instead of where. Both are the same. And there are different ways to write filter conditions. You can write the whole statement in a SQL-like format as a string, you can defer to a DataFrame column using the shortcut dollar sign, or call function with column name or DataFrame name followed by the column name. They all work the same. Execute this, and a new DataFrame is ready with filters applied. Displaying the count triggers an action operation, and you can see that the data has been filtered in the DataFrame. Sounds good? All right, let's check for data completeness. Spark provides a class, DataFrameNaFunctions, which can be used to work with missing data. This class can be used by using na property. So to remove all rows where pickup location or drop location is null, you can use drop function of this class and pass a sequence of columns. Execute this, and this successfully removed the rows. Very quick way to clean up the data. On the other hand, if you want to replace null values with default values, you can create a mapping of default values for columns. Remember, you don't need to provide default for all the columns. And then use na.fill method and pass the mapping. Executing this, replace the null values with default values. Now that the data is complete, let's make sure it is unique as well. Removing data duplication is always a challenge, but here you can do it very easily. To do that, you can use a method, dropDuplicates, on the DataFrame. If you don't provide any column names, complete rows will be used to check for duplicates. If you specify columns, then duplicates will be checked for only those columns. Once you execute this, the duplicate rows are removed, and only one row is retained. Simple and very effective. The last check that we are going to add is for pickup and drop time. Since we are expecting only December 2018 data in this file, any other records which are not in this range should be removed. Let's add the conditions for that and execute the cell. And this makes sure that appropriate date range is applied. Okay, so we have applied a lot of transformation operations, but do we always need to apply each operation individually? The answer is no. You can chain all the operations together. These are all transformation operations and are only executed when the action operation is applied. And remember, it does not execute sequentially. Spark will optimize this plan and then execute, giving you great performance. So, to do data cleanup, you can add checks for data accuracy, add checks for data completeness, for data uniqueness, and for timeliness as well, all in a single statement if you want, and you will have to do the same activity for all the source data. Isn't that making the job of building ETL simpler?

Applying Transformations

Now that we are done cleaning up data, it's time to apply business-specific transformations. Let's apply some transformations to the FHV data. I'm in the notebook ExtractFHVData in which we have already extracted the data for FHV. To clean up the data, I ran the describe command and then added some filters like dropping rows with nulls, removing duplicates, and filtering on dates. It's the same thing that we did with yellow taxi data in previous clip. Let's run a command now, printSschema, on the DataFrame to check the current structure of DataFrame. We have seven columns, but we don't need the last one, Dispatching\_base\_num, in our fact table. So let's go ahead and select only those columns that we need. For this, use select method and mention the column names that you want from DataFrame. Execute this, and see the DataFrame has now been restricted to these columns only. This will also reduce the size of the data frame by processing. Alternatively, you can also use drop method to remove the columns you don't need. In this case, instead of selecting all columns except one, you can also use drop dispatching\_base\_num, and this will give you the same result. And you may have noticed as well the column names needs to improve. You can use the select method again and provide the alias or the new name for any column. Very much like SQL, right? But to do this, you will need to refer it as a column. Let's execute this, and there is an error here, and the reason for this error is that you cannot mix column references and string references here. So to fix it, all the others should be column references as well. And now you can see the new column name. The other way to rename the columns is by the use of withColumnRenamed method where you can provide the original name and the new one. Execute this, and now we have much better-looking names: PickupTime, DropTime, PickupLocationId, and DropLocationId, as well as BaseLicenseNumber instead of dispatching\_base\_number. Along with your original columns, let's introduce some derived columns as well. Let's use the PickupTime to create year, month, and day columns separately. You can use withColumn method on the DataFrame to add a new column, and then you can specify how you want to derive the value of that column. Here, let's use built-in year and month functions on the PickupTime to get the values for new columns, TripYear and TripMonth. Very neat. But this is not the only way to add a new column. Select is a very powerful method. Carefully notice two things here. First, you can specify \* to select all columns in the DataFrame, and second, you can create a new derived column. So you can use another built-in function, dayofmonth, to get the day value and rename it to TripDay in the select method itself. And there is no difference in performance whether you use withColumn or do it via select. Execute this, and now you have three new columns added to the DataFrame. Very useful, right? Okay, let's add some complexity here. We have pickup time and drop time. Now let's calculate the actual trip time in minutes. Again, you can use withColumn to add the TripTimeInMinutes column. To process this, you can use unix\_timestamp function, which returns a number of seconds for the timestamp. You can subtract the seconds of DropTime and PickupTime to get the difference in seconds, and divide it by 60 to get the number of minutes. Here we are using the round function to round the number of minutes. Execute this, and that's it. You have quickly added a new derived column with slight more complexity. Let's look at one more example. Here we are going to use SR\_Flag value. If the value is 1, it's a shared trip, otherwise, it's a solo trip. Add a new derived column, TripType. Use the when clause to check if SR\_Flag is 1. Notice the use of triple equal too for comparison when you are using Scala. If it's 1, mention SharedTrip, else mention SoloTrip. This is the if/else being applied to a column and is also similar to case statement in SQL. And finally, drop the column SR\_Flag as it's no longer needed. Execute this, and you get a TripType field in the DataFrame. Notice we haven't applied any action operations so far, and that's why all commands have executed very quickly. This is what is called as chaining of operations. There are a lot of mathematical, string, and daytime functions available, and of course, we can't go with all of them. Remember, the idea here is to show you the capabilities available that can help you build your ETL pipelines. Now that we are done applying transformations to a single DataFrame, let's see how we can work with multiple DataFrames. If you remember from last module, we also extracted base locations of FHVs. Let's set that again. It's a JSON file, and address and geolocation have a nested structure. You can make this as a flat structure if you want. It's very simple to do that. Let's select some of the columns here and rename them. To extract data from the object in the address field you can use the root field name, Address, followed by the inner field name. So using Address.Building gives you the building name. In the same way, you can extract street, city, state, and postal code and provide an alias for all of them. Assign this DataFrame to a new variable, fhvBasesFlatDF, and let's execute this. And you can see now you have a flat structure. Now we have both, the FHV trip data and the licensed bases data. Let's join these two DataFrames on a common column, BaseLicenseNumber. This is same as joining two tables in a SQL environment. Few things to note here. At first, on first DataFrame, fhvTaxiTripDataDF, which is also called as left-side DataFrame, add a join method. Then provide the name of the second DataFrame, or the right-side DataFrame, fhvBasesFlatDF. Followed by this, provide the columns on which you want to join. There are two ways to provide the join conditions. If columns on which join condition is to be applied have same names, then use Sequence followed by the common column names. Or if they are different, use DataFrame name followed by the column name. After this, provide the type of join. If you do not provide this parameter value, default is an inner join. You can also make it as a cross join, full join, left outer join, right outer join, and more. If you aren't familiar with SQL joins, I would urge you to learn more about it. Let's execute this, and now you have a new DataFrame with columns from both the DataFrames. Of course, BaseLicenseNumber has been specified only once, and how can you select only few columns from this new DataFrame? That's right, by using the select method. And finally, let's create a report. In the report, let's check the total trip time aggregated by city and base type. To do an aggregation, let's first group the data by AddressCity and BaseType. Since we want to get the total trip time, let's use a method, agg, for aggregation, and mention sum of TripTimeInMinutes. Let's execute this. And this gives us the aggregated trip time in minutes across city and base type. And by now you know that you can rename the column using withColumnRenamed function. So let's rename some TripTimeInMinutes to TotalTripTime, and then let's sort the data in ascending order by City and BaseType using the orderBy function. Let's execute and display the results of this report. Very quick and easy to build, right? Let's summarize it by chaining all the operations together. So we started applying the transformations to FHV trip data. We went ahead and did the cleanup of data and applied filters. Then we restricted the DataFrame to only a few columns by using select method. And then we renamed these columns to become more readable using withColumnRenamed method. We then created the right columns to pickup year, month, and day values of pickup time using the built-in functions. And we created another one, TripTimeInMinutes, by using data of two columns, applying a formula, and rounding of the value. And finally, we applied the if/else statement to figure out if it's a shared trip or a solo trip and dropped one of the columns. Great. We then went ahead and joined the FHV trip DataFrame with bases DataFrame using an inner join on BaseLicenseNumber column. And on top of it, we created a report, grouping it by City and BaseType and aggregating by TripTimeInMinutes, followed by sorting of the data. If you carefully notice, we are not just building an ETL pipeline here. We have interactively analyzed the data, cleaned it, applied business-specific transformations, and built a report here as well with just a few lines of code. Interesting, right?

Working with Spark SQL

So far, we have been working with Scala, and by now you very well know that you have different language options to work with. Databricks provide an interactive way to work with multiple languages together, so let's see how to work with SQL along with Scala. But before we do that, let's understand what is an execution context. An execution context is an isolated environment in which the code is executed and state of all variables, objects, and functions is maintained. In Databricks, a new execution context is created on the cluster for every combination of language and notebook. This means if you are writing Scala code in one notebook, it runs in an execution context, and if you write code in another notebook, it's a different execution context. Because they are isolated, objects in one execution context cannot be shared with another one. For example, variable created in one notebook cannot be accessed in another notebook. That also means if you use Scala and SQL in same notebook, the objects created in one language can't be used in another language. So, to work with multiple languages and notebooks, you need to pass around the data from one context to another. Let's see how that works. Back to our Databricks workspace in our notebook, ExtractFHVData, we have already created a DataFrame. Since it's in Scala execution context, you can't use this in SQL. To use it in SQL, you can use a method, createOrReplaceTempView, on the DataFrame, and provide the name with which you want to refer that in SQL. Let's keep it as LocalFhvTaxiTripData. Execute this, and see how quick it is. This creates an in-memory temporary view, which is in the execution context of SQL and is only valid in this notebook only. Think of this like a pointer to the DataFrame. You can now go ahead and use spark.sql method to write a SQL query, and the temporary view, LocalFhvTaxiTripData, can be used as a table. Execute this, and now you're running a full SQL query on an existing DataFrame in Spark. Very useful if you're coming from SQL development background. But now comes the interesting bit. Databricks goes one step ahead and allows you to use magic command, %sql. You no longer need to create SQL as a string and pass to spark.sql. You can directly write the same SQL query in a Scala notebook. Execute this, and you get the same result. Wow! That opens up a lot of possibilities. Now we want to merge data from all types of taxis so that we can prepare aggregated reports across all of them. But as you know, this temporary view, LocalFhvTaxiTripData, is valid in this notebook only. To pass it around to a different execution context or notebook, you can create it as a global temp view using method createOrReplaceGlobalTempView. Provide the name of the global temp view, FactFhvTaxiTripData. Remember, this is our final shape of FHV fact table, and you'll see how to save it in the next module. Once you execute this, this view can now be accessed in any execution context on the cluster. Simple, right? Note that I have created the global temp views for yellow and green taxi data in exactly the same way. Let's now create a new notebook, MergeTaxiData, but this time let's use SQL language. Here we are going to use fact tables of all types of taxis. Let's write a SQL query to merge the data of yellow and green taxis. Notice that it's no longer using SQL magic command. This is because the default language of notebook is SQL. The other thing to note is the use of global temp views. All the global temp views are listed in global\_temp database, so to access the view, you have to use global\_temp followed by the view name. And we are using UNION to merge the data of both the tables. Let's execute this, and we now have a single result set. Let's now write Scala code in SQL notebook. Now you can use %scala magic command to do that. Let's extract the taxi zones data in a DataFrame and then register it as a local temporary view. Why local? Because we only want to use this view in the current notebook. Finally, let's create a report now to figure out number of shared rides grouped by borough and type of taxi. Let's merge all the taxi data where the trip type is a shared trip. Join this dataset with TaxiZones on PickupLocation, group it by Borough and TaxiType, and count the number of shared trips. And finally, sort the data using Borough and TaxiType, and execute this. The idea of showing you this query is that you can write complex SQL statements while the data was extracted in DataFrames. Awesome, right? And let me show you one cool feature here. You can convert the display dataset into a chart instantly by clicking this icon, and you can create different types of charts based on your requirements. This is very useful for business analyst and data scientist.

Handling Corrupt Data

Even though we have done the data cleanup, there are still chances of receiving corrupt data from the source. It could be because of syntax errors or schema mismatch. Let's see how we can handle that. Spark provide out-of-the-box support to handle corrupt data. The text file formats like CSV and JSON has built-in support to handle corrupt records in a file. There is slight difference in the way they handle it and that we'll see in just a moment. There are three parse modes which are supported, and you apply them while reading the data. These are Permissive, DropMalformed, and FailFast. But along with this, Databricks has a unique feature. You can specify a location, and Databricks will automatically store the corrupt records along with a failure reason to that location. Let's see how that works. Just to see the effects of different parse modes, let's create a sample notebook to try it out. First, let's check them on a JSON file. Let's see what's present in RateCodes.json file by using the head command. And you can see that one of the records is not correctly formed. Let's read this file in a DataFrame just like we have done it so far. Execute the cell, and notice few things. At first, a new column with the name \_corrupt\_record has been created. Second, the bad record has moved to the new column, and the values of RateCodeID and RateCode are null. This is what is called as before permissive mode. You can even explicitly specify the permissive mode by using option method. Specify the key as mode and value as Permissive. Execute this, and this gives the same result. Next, let's change the mode from Permissive to DropMalformed. Execute the cell, and now you can see that the corrupt record has been removed from the DataFrame, so it drops all the records which are not in the correct format. Now let's change the mode to FailFast. Once you execute this, you can see that it threw an error. So on a JSON file, Permissive stores the corrupt records in a new column, DropMalformed removes the row, and FailFast results in a failure. Depending on your requirement, you can configure the mode. All right, let's see another option. Instead of specifying mode, you can add another option, badRecordsPath, followed by a location. This option is provided by Databricks. Let's execute this and notice that it has removed the bad record. But the information of this bad record is now available in the badRecordsPath location. It creates a new file in a subfolder and contains the location of file with errors, the exception reason, and the corrupt record. This is very useful to handle and track the corrupt data in source files. Let's do the same for a CSV file. The only difference in handling a JSON file and a CSV file comes during permissive mode. The behavior of DropMalformed and FailFast, that means the same, and so is the behavior of badRecordsPath. Now I have the same records, but this time in a CSV file. And you can notice that it has two corrupt records this time. Let's try to read this in a DataFrame. And specify mode as Permissive since this is the only difference as compared to handling JSON files. Once you execute this, notice that it does not create any corrupt record column while reading the CSV. It only goes ahead and specifies the missing values as null. Of course, adding a string value to RateCodeID may not be something you want, but depending on your requirements, you can use any of these three options or the badRecordsPath that Databricks provide.

Summary

In this module, we covered a lot of details of the second layer, the transformation layer. We started by understanding the dimensional model for taxi data, which we used in our demo. We then saw some of the commonly used transformations, if they are available is SSIS and Informatica, and how you can get the same functionality in Scala and Spark SQL. We then started by analyzing the data and used a built-in method, describe, to get the summarized view of a column in a DataFrame. Once analyzed, we applied the data quality checks like completeness, uniqueness, timeliness, and accuracy and cleaned up the data using DataFrameNaFunctions like drop and fill, used dropDuplicates to delete the duplicate roles, applied filters, and more. Followed by this, we applied the business transformations using common methods like select, withColumn, withColumnRenamed, et cetera. We also saw how to join two different DataFrames and used groupBy, aggregate, and orderBy functions to build a report. Then we saw the great features of working with multiple languages. We saw how to use Spark SQL, along with Scala. We can use spark.sql method or use %sql magic command to write the SQL queries. To pass around the data from Scala to SQL, we registered the DataFrame as a temporary view. You can create a local view to use it in the same notebook or use global view to access it in any session in your application. Global views are accessible using global\_temp database. And finally, we saw how to handle the corrupt records in JSON and CSV files by using different mode types, Permissive, DropMalformed, and FailFast. And we also saw how to use badRecordsPath to store the corrupt records in a separate file. This is just the beginning of applying transformations. You can build the same level of complexity as in other ETL tools with less development effort and get the benefits of scale and compute that Databricks offer. In the next module, we'll see how to load the data into the destination.

Loading Data

Module Overview

Hi, and welcome to this module on Loading Data. Now that we are done applying transformations to the source data, it's time to load the process data to the destination. We'll start by loading the process data to the Data Lake in CSV format. We'll see various options which are available and different modes for writing the data to the destination. We'll then understand what happens behind the scene when we save a file to the underlying store. You will also learn about partitioning in Spark and how you can control that. Next, we'll understand what is Apache Parquet format, what are its benefits, and how we can load the data in this format. Followed by this, we'll understand the concept of databases and tables supported by Databricks. You'll then see the differences between managed and unmanaged tables, how you can load the data in both types of tables, and where you should use them. Sounds good, so let's get going.

Loading to Files

Let's start by loading our transform data into files. There are multiple file formats in which we can load the data depending on our requirements, and we'll see some of those options here. In our Databricks workspace, I'm in the notebook ExtractYellowAndGreenTaxiData. We have already created two DataFrames, one for yellow taxi and other for green taxi data, and we have applied transformations on both of them. These DataFrames have the final shape of our fact tables. Let's first load the green taxi DataFrame as a CSV file to our Data Lake. The process of saving a file is as simple as reading a file. On the DataFrame, greenTaxiTripDataDF, apply the write property. You can also specify some options just like we did while reading the files. Let's add the option with header as true, and let's also add the dateFormat. All the fields with timestamp data type will be written in this format. Next, specify the CSV method, and provide the destination path as a parameter to the CSV method. Let's create a folder, DimensionalModel, and a subfolder, Facts, in the mounted Data Lake, and keep the name of the file as GreenTaxiFact. And that's it. Write property on the DataFrame is an object of class DataFrameWriter and support a lot of methods that allow you to apply various options on the DataFrame and help you save the DataFrame in different formats like CSV, JSON, Parquet, and more. Let's execute this, and this will save our transformed green taxi data to the Data Lake. Very simple, right? Now let's re-execute this command, and as expected, it threw an error. This is because the file GreenTaxiFact.csv already exists. To overwrite the file, let's add another method of DataFrameWriter class called mode. In the mode, you can specify the behavior when data already exists. Let's specify SaveMode.Overwrite. There are four types of modes you can specify: Append mode. If the data already exists, the new DataFrame contents will be appended to the existing data. If it does not exist, a new dataset will be created. ErrorIfExists mode. If the data or file already exists, it will throw an error. Third is the Ignore mode. If the dataset already exists, the write operation will not affect the existing dataset. This means the DataFrame that we have created will simply be ignored. And the last one is the Overwrite mode. That will overwrite any existing dataset. So here, we are specifying the Overwrite mode that will overwrite the file we previously saved. Execute this, and now this command works successfully. Let's now switch over to the Data Lake account to verify if the file has been successfully saved. Let's open the folder, DimensionalModel, Facts. And that's strange. Instead of GreenTaxiFact.csv file, there is a folder with the same name. Inside this folder, you can see huge number of small files that have been created, starting with keyword part. If you open any of the part files, you can see that the data is there. But what's going on? Let's understand why multiple files have been created when we saved the DataFrame. As we discussed in the second module, RDDs or DataFrames are broken into partitions, and these partitions are distributed and stored in the memory of multiple nodes, which are there in the cluster. Each node can store multiple partitions, and any processing that you do happens on these partitions. This helps in doing the distributed data processing and helps to achieve parallelism. That's why, while saving the file, a folder is created instead of file. Like in our case, GreenTaxiFact.csv folder was created, and data in each process partition is written as a separate file inside this folder. And since default number of partitions in Spark is 200, so 200 partition files or part files are created inside this folder. That's how you can process terabytes and petabytes of data in parallel, but the result is written as multiple partition files. Let's understand this partitioning in a bit more detail. The first important point is, how do you consume this data if it's written in multiple files? If you want to consume this in Spark or any other tool with Spark connector, you can simply specify the name of the folder, like GreenTaxiFact.csv, and it will read all the partition files. The default number of partitions in Spark can also be changed by changing the configuration setting, spark.sql .shuffle .partitions. Or if you want to change the partitions of only one DataFrame, you can use coalesce method or repartition method and specify the new number of partitions as a parameter. But be careful as changing that results in the movement of data between partitions. As a rule of thumb, use repartition method when you are increasing the number of partitions and coalesce method to decrease the number of partitions. This is because of the shuffling behavior. Remember, optimal data partitioning can help you achieve best performance for your Spark jobs. If you are very new to Spark, I would suggest you read more on Spark's data partitioning. Switching back to our notebook, let's first check the number of default partitions on the cluster. To do that, you can use spark.conf .get method and get the value of any Spark configuration. Here, let's specify spark.sql .shuffle .partitions. Once you execute this, you can see that the current default partitions are set to 200. You can change this value by using spark.conf .set method, provide the config key, and specify the value as 10, which means going forward, all the new DataFrames would get default 10 partitions if number of partitions are not specified. Remember, even after changing this property, the DataFrames that are already created won't be affected. This means our DataFrame, greenTaxiTripDataDF, will still have 200 partitions. Let's check that by using a command on DataFrame, rdd.getNumPartitions. Once you execute this, you can see that our DataFrame has 200 partitions. You can change the partitions of existing DataFrame by using coalesce or repartition methods. Since we want to decrease the number of partitions, let's use coalesce command and specify the number of partitions as 1. Execute this, and now you can see that our DataFrame only has a single partition. Let's save this DataFrame back to Data Lake by executing the same write command. Switch back to the Data Lake account, and now you can see only a single part file. Let me reiterate this as a warning. Creating a single partition is fine with smaller datasets, but should not be used with larger datasets. This is because it not just results in the movement of data, but all data coming to one partition can become a bottleneck, could take a lot of time to save, or even lead to memory exceptions. Now let's see how we can store this data in a different file format. Apache Parquet file format is one of the widely used formats and provides great performance. Unlike CSV or JSON, which are role-based formats, Parquet stores the data in a columnar format. Just like JSON, it can store complex data structures, as well as nested data. The great thing about Parquet is that it also stores the schema of the data in the file itself. So when you want to read data from Parquet, you don't have to define or infer the schema. It can directly be read from the Parquet file. It also supports efficient compression and encoding. That's why Parquet files are much smaller in size than CSV files. Because of this, Parquet files are binary files, and it may take more time to write to Parquet than to CSV files. But reading from Parquet is extremely fast, especially more when you are accessing only a subset of columns. Let's see how you can save the DataFrame to the Parquet format. Back to our notebook, writing to Parquet is very similar to CSV file. On the DataFrame, use the write property and specify the different options. Let's specify the mode as Overwrite. And on top of it, call the parquet method and pass in the destination path. Execute the command, and this saves the data as a Parquet file. Let's switch to the Data Lake and check the size of CSV and Parquet files. The only part file of CSV is 60+ MB. And let's check the part file of Parquet, and you can see that it is less than 15 MB. You can also compress the CSV files, but Parquet does that by default. Let's switch back to the notebook, and let's now try to get the count of distinct values of PickupLocationId and DropLocationId from both the files. At first, let's read back the CSV file in a new DataFrame and write the query to get the count of distinct values. Execute this, and notice the time here. Let's write the same code, but this time read from a Parquet file. Execute this, and now you can see reading the data from Parquet is much faster. So whenever you want to read the underlying files multiple times, it's worth storing them in Parquet format. They provide great compression and extremely good read performance. Awesome, right?

Working with Databricks Tables

So far, we have only saved the data directly to files. Databricks also has support for databases and tables, but as you saw in module two, this is very different from a relational database. Let's understand what are these tables and how they can be used. In Databricks, a table is just a representation of a dataset while the dataset is stored at an external location. For example, you can represent a file stored in Data Lake as a table in Databricks. You can either use default database or create new ones, which will contain the tables. Whenever you create a Databricks workspace, it has a central Hive metastore. This metastore is used to store the metadata, or the schema of the tables, and as you know, it's nothing but the schema of the underlying dataset. This means it has columns, and columns have data types. This metastore is accessible by all the clusters. The underlying data can be in any format. It can be a CSV, JSON, or Parquet file, or it could reference an RDBMS table or NoSQL collection. Think of it this way: Any source using which you can create a DataFrame can become a table, so you are kind of registering the DataFrame as a table. Once a table is defined, you can use it like a SQL table or can even use it as a DataFrame in Scala without providing credentials. There are two types of tables, a managed table and an unmanaged table. In a managed table, both the schema and the data is managed by Spark. Because it is managed by Spark, data is stored in DBFS, and it's in Parquet format. But remember, if you drop the table using drop command, both the schema as well as the data will be removed from DBFS. Also, if you delete the Databricks workspace, DBFS will be deleted, and you will lose the data of the managed table, so you should only use it to persist any staging data during your ETL workloads. On the other hand, in an unmanaged table, only schema is managed by Spark, which is stored in Hive metastore, while the data of the table is stored in an external location. For example, if you have a file in your own Data Lake that can be registered as an unmanaged table. Since you are only referencing an underlying dataset, the data can be in any format, be it CSV, JSON, Parquet, RDBMS table, et cetera. Now, if you drop an unmanaged table, only schema is removed from Hive metastore. The data is not removed from the underlying source. That's why unmanaged tables are used to persist the processed data, like dimensions, facts, and reports. Makes sense, right? Let's see how we can work with both types of tables. Back to our Databricks workspace, let's first use a statement to create a database, CREATE DATABASE IF NOT EXISTS, and provide the name of the database, TaxiServiceWarehouse. Execute this, and it will quickly register a database with Hive metastore. To check if the database has been created successfully or not, on the left-hand side, go to Data tab, and you can see that database has been created successfully. Next, let's use the same DataFrame, greenTaxiTripDataDF, use the write property, specify mode as Overwrite, and instead of storing it as a CSV or Parquet, you can use another method, saveAsTable. As a parameter, provide the name of the table, TaxiServiceWarehouse.FactGreenTaxiTripDataManaged. Execute the cell, and this stores the DataFrame as a table. This is what is called as a managed table, as we are not specifying any external location to store the data. Because of which, it is stored in DBFS. To verify its creation, you can go back to Data tab and see that table has been created in the database. If you click on the table, you can see the table metadata, as well as its data. You can also use command SHOW TABLES to see the list of all tables in a database. Now that the table is registered, let's run a simple select query. Execute this, and this gives us 10 records from the table. To check the details of this table, let's run another command, DESCRIBE EXTENDED, followed by the table name. Once you execute this, it shows you the list of columns. You can also see detailed table information. It tells us the type of table is managed, and underlying data format is Parquet. It also shows the underlying location of files on DBFS. And now you know if you drop this table, the data will also be deleted and cannot be restored back. That's why we're not going to keep our GreenTaxiFact as a managed table. Makes sense, right? To store our DataFrame as an unmanaged table, we are going to use the same command, but this time, let's add an option of path, and provide an external location. If you don't specify the format, it stores the data as Parquet, but you can specify if you want to store it as a CSV, and keep the name of the table, FactGreenTaxiTripData. Let's execute this, and this will create a new table. Let's run the DESCRIBE command again to check the properties of this table. You can see the column names, but this time see that the type is external, which means it's an unmanaged table. Underlying format is Parquet, and notice the location. It points to our Data Lake. Let's now try to drop this table. Once you execute this, the table is dropped. Let's switch over to the Data Lake and see the status of the underlying file. And you can see that dropping this unmanaged table hasn't removed the underlying file. Great. Let's just recreate this table directly from the underlying file now. You can directly do that in SQL. Specify CREATE TABLE command, mention that underlying file format is Parquet. Here you can mention CSV, JSON, and even JDBC to refer to an RDBMS table. Since Parquet contains a schema, you don't need to explicitly define the schema of the table. In the OPTIONS, specify the path of the file. Once you execute this, it will register this table again as an unmanaged table in the Hive metastore. And that's how you can save all the dimensions, facts, and reports not just as underlying files, but you can also register them as tables for better control and faster development. Sounds good?

Summary

In this module, we covered various ways of writing the data to a destination. We started by loading the data in CSV format and applied various options like header and dateFormat. And we also saw different types of SaveMode, Append, ErrorIfExists, Ignore, and Overwrite modes. We then saw that saving a DataFrame as a file actually creates a folder with the same name. The RDD or DataFrame partitions are written as separate files inside that folder. The number of part files depends on the number of partitions of the DataFrame. And that's where we can decide and change the default partitions in Spark or change the partitions on a single DataFrame. We can use the configuration setting, spark.sql .shuffle .partitions, to change the default partitions, or use coalesce or repartition methods to change the partitions of a DataFrame. We then saw the benefits of Parquet format over CSV, its properties, and how we can load the data in Parquet format. We also saw that Parquet provides higher read performance and great compression. Databricks also has support for databases and tables. Table is just a representation of the underlying data, and its metadata is registered in Hive metastore. Then we understood the differences between managed and unmanaged tables, where the underlying data is stored for both, and what's the impact of deleting these tables. In the last three modules, we have seen how we can extract, transform, and load the data, and we have seen how easy it is to explore the data and quickly build the ETL pipeline using Azure Databricks. In the next module, we'll clean up some of the code, make it production-ready, and orchestrate the end-to-end pipeline using Databricks Jobs and Azure Data Factory.

Orchestrating ETL Pipeline

Module Overview

Hi, and welcome to this module on Orchestrating ETL Pipeline. Now that we are done building the extract, transform, and load steps, it's time to orchestrate the end-to-end pipeline. Since we wrote a lot of code to explore the data, we'll start by cleaning up and organizing the code. We'll then add parameters to the notebook so all the hard-coded values can be made configurable. We'll also see how to return the values from a notebook just like we return the values from a method. And then we'll work on building a workflow by calling one notebook from another one, pass the parameters, and accept the return values. After we are done creating workflow for one notebook, we'll orchestrate our notebooks for dimensions and facts. We'll then go out and schedule the workflow using Databricks Jobs. And finally, we'll see how to orchestrate an end-to-end pipeline and schedule that using Azure Data Factory. Sounds good? So let's get going.

Setting up Workflow

We'll start by setting up an end-to-end workflow and make our pipeline production-ready. Since we wrote lot of code to understand the features of Databricks, let's clean up some of the code, organize all dimensions and facts in separate notebooks for better control, add relevant comments, and log information. Switching back to our Databricks workspace, I have created another root folder, PluralsightDemoProd. I have created two subfolders, Dimensions and Facts, so we can keep separate notebooks for separate entities, and a Reports folder for generating any reports. In the Facts, there are three notebooks, one for each fact. If you open ProcessFactYellowTaxiTripData, you can see that important description has been added using %md command. Then I have added some custom log information using println command throughout the notebook. Once the notebook runs, this information will show up in the log, And then we have the same ETL code that we wrote in previous modules. At the end, I'm registering the DataFrame as a global temp view so it can be used in other notebooks. Remember, every time this view is accessed, more transformations will be evaluated again. If your transformations are complex and require lot of time, consider persisting the output as a table just like the below statement. Next, I'm saving the DataFrame contents to our fact table by using saveAsTable and providing the external path for the data. And by now you know that it's an unmanaged table. Also, this time, I'm appending the contents to the existing dataset, And the notebooks of other dimensions and facts I have created exactly follow the same pattern. Now, since the same workflow will run for each month's file, we need to get rid of hard coding this month value. Let's pass it as a parameter. To create a parameter for the notebook, we are going to use another Databricks feature called Databricks Widgets. To create a widget, let's run a command, dbutils.widgets .text. This creates a text-based input. Provide the name of the widget, ProcessMonth, a default value, and a display label value. Let's execute this, and you can see a widget has been added at the top of the notebook. This will act as a parameter to the notebook. There are four types of widgets you can define: Text widget that we just created, allowing text as an import. You can create a dropdown widget where you can select a value from the list of available values. Next is a combobox widget, which acts as a combination of text and dropdown widgets. You can either select a value or type in a new value. Last one is the multiselect widget where you can select one or more values from the list. Let's change the value of the widget to 201812. To extract the value of this widget, let's use another command, dbutils.widgets .get. Provide the widget name, ProcessMonth, and assign this value to a variable. Once you execute this, you can see that it picked up the new value from the widget. Let's now go ahead and use this variable in the file name. Either you can concatenate this variable with the file name, or you can also add s in front of the string, which is called string interpolator. Because of this, you can directly refer the variable in string now using dollar sign. This will replace the variable name with its value. Interesting, right? The next thing we are going to do is to set up a return value from this notebook. At the end of the notebook, let's write a command, dbutils.notebook .exit, and as a parameter, pass in the return value. You can pass in variable value or any other value, but not DataFrames. Here, let's specify Success, which will act as a return value of this notebook. And that's how we can add parameters and return values in all the notebooks. Easy, right? Now that we are done setting parameters and return values, let's see how we can use them. Let's create a new notebook, OrchestrateTaxiServicesWorkflow. Here also, let's create a parameter for processMonth, and in the next step, extract this value in a variable. Let's execute this, and the processMonth variable is ready. Now let's invoke a dimensional fact notebook from this notebook. First, to pass the parameters, let's create a map of parameter names and parameter values. The name of parameter is ProcessMonth, and the value is coming from variable. Second, let's use a command, dbutils.notebook .run. Pass in the path of the notebook. Pass the time of values in seconds. Zero specifies no timeout value. And then pass the map of parameters. Finally, assign the return value from the notebook to the variable status. And third, let's check the return value to see if the notebook executed successfully. Of course, you can and should add exception handling in this code. And that's it. Let's execute this. And you can see a notebook job has been created. Let's click on this. All the cells in the notebook are now getting executed sequentially. Once complete, it will send the return value back to the orchestration notebook. And you can see that it returns Success, and we are displaying the right message. That's awesome. Let's now update this code to run all notebooks. Let's invoke all notebooks for dimensions first. And then invoke all the fact notebooks. All of this will run in sequence. But with slight code change, you can invoke those in parallel, which don't have any dependencies. Of course, you should go ahead and do exception handling, check for return values in all cases, and then take appropriate action. But I'm sure this must have given you ideas on how to do that, right?

Scheduling with Databricks Jobs

Now that we are done setting up the workflow, it's time to schedule and run the workflow using Databricks Jobs. Back to the Databricks workspace, setting up a job is very intuitive and a straightforward process. On the left-hand side, there is a Jobs tab. Click on it to access the list of jobs. Click on Create Job to set up a new one. Let's understand and fill up the properties here. Provide the name of the job. Let's specify TaxiServiceMonthlyJob. There are three ways to run the workflow or the task. First, you can select the option of setting JAR file. Here you need to upload the JAR file, which contains your Spark code. Provide the name of the main class and the optional arguments, and once the job executes, the main function will be invoked. Second option to run the workflow is by using spark-submit. Spark-submit is a shell script that allows you to run Spark code. You need to provide the path of the JAR file and other arguments as parameters to the spark-submit command. And the third option is to select a notebook in the Databricks workspace. And this is what we are going to use here. Let's select the notebook OrchestrateTaxiServicesWorkflow. Remember, you can only extract one notebook from here. But as you know, this notebook is our master notebook and is internally calling other notebooks, so selecting this will execute the whole flow. Next, add the parameters for the notebook in the key-value format. Since we have created ProcessMonth as the notebook parameter, let's add ProcessMonth as key and 201812 as the value and confirm the information. You can also add dependent libraries if you are using them in the project. Let's now provide the cluster information. Wow! This means every job can have a different cluster configuration. Click on Edit, and you can see that there are two options. Select the Existing Interactive Cluster option, and choose one from the list. Or you can select new job, or Automated Cluster option, and provide the same information as you did while creating an interactive cluster: pool information, Databricks runtime, autoscaling option, and configuration of worker and driver nodes. The only difference here is that there is no auto-termination option. Why? Because automated clusters are created and terminated with the job. Next, let's specify the schedule to run this job. Select the frequency, time window, and the time zone. There are a few more options available. In the Advanced section, you can specify email alerts. Here you can specify emails to which you want to send out information on start of job, successful completion of job, or on job failure. You can then specify number of maximum concurrent runs, which means number of runs for this job that you can invoke in parallel. Next, specify the timeout value of the job, and then specify the number of retries in case of job failure. Finally, specify the job permissions, just like we specified permissions for clusters and notebooks. And that's it. Our job is now fully configured. Even though the schedule is defined, you can trigger the job manually as well. Let's click on Run Now to invoke it. And you can see the new job has triggered. You can monitor this job run by clicking on Run ID. It will create an automatic cluster, run the job, and terminate it as soon as it is done. Simple, right?

Orchestrating with Azure Data Factory

So far, you have seen how you can orchestrate the ETL pipeline by invoking multiple notebooks from one notebook and scheduling them using Databricks Jobs. Now let's see how you can orchestrate and schedule that using a great tool, Azure Data Factory. Before we move to Azure Data Factory, there is one thing that we need to pick up from Databricks in order to connect the Databricks with Data Factory, and that is the access token. Back to Databricks workspace. On top right, go to Account, User Settings, and there you see the option of Access Tokens. Click on Generate New Token. You can add a description and the validity of the token, and clicking on Generate will generate a new token. Copy and save this value as you won't be able to retrieve it again. Next, create an instance of Azure Data Factory and launch the authoring window. It will bring you to this page. Let's get started by authoring a new pipeline. To do that, click on Connections first, and let's create a new linked service. A linked service in ADF is a connection to the underlying source or compute service. This will allow you to set up a connection to Databricks. Click on New linked service, and go to Compute tab. And there you can see the option to connect to Azure Databricks workspace. Select Azure Databricks, and here you can provide the connection properties. You can either select Databricks workspace in your own subscription or in any other subscription. Let's select our Databricks workspace here. Next, select if you want to use a job cluster or an interactive cluster or if you want to use an existing instance pool. This is exactly the same process as you saw while setting up the Databricks job. This time, let's select Existing interactive cluster. And now you can see the option of Access token. Paste the value of access token that you copied from Databricks workspace. This will allow ADF to access Databricks workspace. And now it shows up the list of all interactive clusters. Select one of them, and that's it. This has now set up our underlying compute using existing interactive cluster. Now that the linked service is ready, click on New pipeline to set up a new one, change the name of the pipeline, TaxiServicesPipeline. See the category of Databricks in Activities. Select Notebook activity and drag onto the panel. Change the name of the activity to OrchestrateTaxiServices. The properties that you saw for Databricks Jobs, very similar properties exist here as well, Timeout, Retry, and Retry interval. Next, go to Azure Databricks tab, and let's select the linked service we created. Click on Test connection to see if it works. Yes, it does. All right, let's go to Settings tab. Here you can select the notebook you want to invoke. Click on Browse, and select the same, OrchestrateTaxiServicesWorkflow. In the parameters, add the name of notebook parameter, ProcessMonth, and add the value here, 201812. This means parameter value will now be passed from ADF. And finally, you can define a trigger using which you can schedule the pipeline run. Publish all the changes, and you're done. You can now trigger the pipeline immediately or on a schedule. Easy, right? One last thing. If you're coming from ETL development background, you definitely know that you can define the control flow. Instead of calling one single orchestration notebook in ADF, you can build a full workflow in ADF and call each Databricks notebook separately. For example, you can create a master pipeline to process the dimensions first and then process the facts. In the dimensions pipeline, dimensions can be processed in sequence or in parallel by calling individual notebooks of taxi zones, payment types, et cetera. In the same way, you can invoke notebooks for each fact separately in the child pipeline for facts. This gives you much more control over your workflows. Sounds good?

Summary

In this module, we covered various ways of orchestrating and scheduling our end-to-end ETL pipelines. We started by cleaning up all the exploratory work and organized all the entities in separate notebooks so they can be better managed. We then went ahead and parametrized the notebook by using Databricks widgets. There are four types of widgets available, text, dropdown, combobox, and multiselect. And we can extract data from widgets using databricks.widgets .get method. Then we saw how to return values from a notebook after its execution is complete using dbutils.notebook .exit method. You can return variable or a text value, but not a DataFrame. After learning how to parametrize and return values from notebook, we built an end-to-end workflow. We called one notebook from another one using dbutils.notebook .run method. We can pass in parameters through this method, and we also accept return values. And in the same way, we orchestrated all the notebooks for dimensions and facts. You then saw how to schedule the orchestration notebook using Databricks Jobs. You can either use an existing interactive cluster or use a job cluster. Job cluster, which is also called as automated cluster, starts and terminates with the job. And we also saw various phase of submitting the job using notebook, using JAR files, and using spark-submit option. And finally, we saw another way of building the workflow using Azure Data Factory. We saw how to call the single orchestration notebook or build complete orchestration inside ADF by calling each notebook separately. This way, we have completed building, orchestrating, and scheduling our end-to-end ETL pipeline. In the next module, we'll discuss other interesting aspects that can help you build better pipelines and automate its deployment.

Building Better Pipelines on Databricks

Module Overview

Hi, and welcome to this module on Building Better Pipelines on Databricks. We'll have a look at couple of interesting features that will help us build automated and reliable pipelines. We'll start by learning about Databricks APIs. We'll see for which services APIs are available. We'll then go ahead and invoke the APIs using PowerShell. First, we'll read the data using an API call and then trigger an action using it. Next, we'll learn about another component, Delta Lake. We'll see why is it required, how you can use it in your applications, and how does it work behind the scenes? Then we'll see the features it supports, followed by the use cases it enables. Sounds good? So let's get going.

Using Databricks APIs

Databricks support variety of APIs to accomplish various tasks. While you may use ARM templates or Azure SDK to deploy new Databricks workspace in Azure, you will need Databricks APIs to work with different components of Databricks. Databricks provides REST APIs to access Databricks services like clusters and jobs as an alternative to the UI. You can use them to access the services programmatically or even to automate end-to-end continuous integration and continuous deployment pipelines, or CI/CD pipelines. The APIs are available for workspace, like creating folders and importing/exporting notebooks. It's available for clusters, as well as pools. So you can create, update, terminate, or delete them. It also has support for jobs where you can create, update, or delete jobs, as well as handle its execution. Then there are DBFS APIs to work with the file system and also the library APIs to install or uninstall them on a cluster. You can also use REST APIs to create execution context, execute the commands within that context, and much more. Let's see a sample of what you need to call a REST API and how you can invoke them using PowerShell. You can collect all the parameters to invoke the API in this variable, info. You'll need the URI of the API method and the authorization information in the headers. The location in the URI is the deployment region of the Databricks. For example, our workspace is in East US 2, so location value will be eastus2. Then specify the endpoint. Endpoint is the URL of the API you want to invoke. For example, to get a list of all the jobs, the URL is 2.0 /jobs/list. The endpoint information is available in the Databricks documentation. Next, you need to add the access token to enable bearer authentication. In the previous module, we created a new access token in the Databricks workspace by going to access token in the user settings. And finally, you can call the REST API by using Invoke-RestMethod function. Pass in the HTTP method like GET, PUT, POST, or DELETE, and this will invoke the Databricks API. Easy, right? Let's see an example how we can get a list of all the jobs in our workspace. I'm in the PowerShell app on my machine. Let's copy the same command we just saw and replace the values of location, API endpoint, access token, and HTTP method. Let's execute this. And now you can see the list of all the jobs returned in the JSON format. Let's try one more. Let's now try to create a new group for the users. Location remains the same, eastus2. Specify the endpoint for it, 2.0 /groups/create. Add the access token to the header. Because this API requires you to pass the name of the group that you want to create, let's pass the group name PluralsightUserGroup as a parameter in the body. Since these values should be in JSON, let's convert that into JSON using method ConvertTo-Json. And this is a POST method, so specify POST as the HTTP method. Let's execute this. And you can see the group has been created successfully. There are even open source SDKs available on top of Databricks APIs that you can use to do automation. Sounds good?

Understanding Delta Lake

Let's see another interesting component, Delta Lake. But before we jump into it, let's understand why we need it. As you have seen in this course, we have been using Data Lake, not just to consume the data, but also to store the process data. So Data Lake is a central repository that can store all types of data, be it structured data or unstructured data, in its native format. And you can store vast amounts of data in a data lake. And with the growing data volumes, data lakes are becoming increasingly important. But using Data Lake brings its own set of challenges. First, if multiple processes are trying to read and write to the same file at the same time, there are chances that readers would see dirty data since data writing is not complete yet. Next, a very common problem is of data inconsistency. If a job fails while writing data, the data is left in a partial state. You have to work around to ensure that data is only written if job succeeds. Then, as you start appending the data, the data is scattered across increasing number of files, thus slowing down the performance over a period. And finally, the biggest of all is the update of records in a file. Even for a single record update, you may have to read an entire dataset, update the data, and save it back. This is where typical data warehousing tasks, like slowly changing dimensions, becomes a challenge. And this is where Delta Lake steps in and was earlier called as Databricks Delta. Delta Lake is an open source storage layer that brings reliability to data lakes. Let's see how it works. To work with Delta Lake, first you need to save the data in delta format. To save the DataFrame, like our yellowTaxiTripDataDF, use the write property, specify the format as delta, and provide a location to save this. As simple as that. This is similar to saving data in other formats, but now the file is in delta format. Let's see what happens behind the scenes. Delta Lake is a component which is deployed on the cluster. In Databricks, it is deployed by default via newer versions of Databricks Runtime. If you are saving the DataFrame in delta format, it gets stored on the storage in Parquet format only. But along with the file, delta stores the transaction log as well. And this is where the magic starts. Let's see what is this transaction log. Transaction log is also called as Delta log, and it's an ordered log of every single transaction that has been performed on the Delta table since it was created. So if you want to read the data from the Delta table, transaction log will be checked, and it will give you the latest view of the table. Transaction log contains all the commit information of transactions. It contains the operation that has been performed, whether it's an insert, update, delete, schema change, et cetera, predicates that have been used to update or delete the records, and all the partition files that have been affected because of the operation. Let's see different scenarios to understand this. As we save the Delta table to the lake, it stores partition files, and transaction log files are stored in \_delta\_log folder. Let's assume two part files are stored and one log file, 000.json, is stored corresponding to that. Now there is a writer process and a reader process, and both of them want to work at the same time. Once the writer process starts, it starts writing the new changes to part file 3. On the other hand, reader process first reads the transaction log. It only reads 000.json log information. As soon as writer process finishes writing to part file 3, it then adds a new file to the log, 001.json, that contains information of part file 3. While on the other side reader has only read 000.json log file, it only gets to know about part file 1 and 2. That's how multiple processes can work simultaneously, and reader will always get the committed data and not the dirty data. Awesome, right? Let's check one more scenario. In this case, we have a writer and a reader process. Writer process starts to write the partition files to the lake, but this time there is an error in the job, which leads to incomplete part file 4 being added to the folder. Because of this failure, Delta Lake module does not write any information to the log. Now when the reader process wants to read the data, it reads the transaction log. Remember, the log does not have any information of part file 4, which is incomplete. That's why, based on log, reader process only reads part files 1, 2, and 3. Makes sense, right? Now this is where comes some major benefit. You can update even a single record in Delta table, and it's extremely fast. We again have two processes, writer and reader. The writer process wants to update a record, which was present in part file 1. Instead of updating that record, it creates a new file, part file 5, and stores updated data in that. It then goes ahead and updates the log by writing 002.json file. This new log file also knows that part file 1 is not needed anymore. That's how the information is updated in the Delta table. On the other hand, when reader wants to read the data, log file is read by Delta Lake, and it only reads part files 2, 3, and 5. Part file 1 is no longer required, and part file 4 is incomplete. So transaction log is the magic behind the scenes that allows Delta Lake to perform ACID transactions on files, and it uses serializable isolation levels. Just to reiterate, underlying file format for Delta is nothing but Parquet, and the log information is stored in \_delta\_log folder. Data in Delta table is only considered written when transaction log is updated, otherwise not. And this log also provides the full audit trail of all the changes that have happened. And that's how it helps in doing MERGE, UPDATE, and DELETE operations on the underlying files. Not just that, but Delta Lake has some other great features too. It enforces the schema on the Delta table that helps in preventing any bad data being added to the table. Because of transaction log, you can get snapshots of older versions of the data from the Delta table. You can access by a version number or snapshot at a particular instance of time. This is called as time traveling. Other great feature is Z-Ordering where you can sort the data in the files based on multiple columns. This allows for quick retrieval of the data. By default, Delta table also collects statistics in form of minimum and maximum values of each column for each part file. Wow! This is making it very close to SQL environment. This speeds up query performance as part files can be completely skipped by just looking at the statistics. And finally, you might be wondering, what about unused files? Delta Lake allows you to do garbage collection by using VACUUM command. This is just a glimpse of what Delta Lake can do. It's very new, and a lot of features are being added to it. Now that you know that Delta Lake allows you to update the data, let's see how you can write the code for it. Let's write it in SQL. You can use MERGE statement and specify the target table, which in this case is DimTaxiZones. Specify the source table, which contains the updates. Here it is StagingTaxiZones. And join both the tables on their keys. If a record only exists in the target table, then update the values you want. And if it does not exist in the target, you can insert the record. And of course, you can also do deletes here based on requirements. And you can do this in Scala as well. Easy, right? But let me share a warning. Reading underlying delta file directly may cause issues, and this is because, you know, delta table stores multiple versions of records. So to use the file directly, you'll need to export it in a different format like Parquet or CSV. So you have seen how useful Delta Lake is, and it can help you build your data warehouse in the data lake reliably. You can implement slowly changing dimensions in the lake. It also helps to enable change data capture scenarios. Using time travel, you can restore the data back in case of failures. And Delta Lake can also act as a common storage for batch and streaming data. Sounds good?

Summary

In this module, we covered two features. Databricks has support for REST APIs for all of its services, be it workspace, clusters, pools, jobs, DBFS, libraries, et cetera. Then we saw how to invoke the APIs using PowerShell by passing URI of the API, bearer token in the authorization header, and the HTTP method while invoking it. And that's where we saw samples to read and write the information using APIs. Next, we looked into the Delta Lake component. We understood the need for it, and we saw how easy it is to save the DataFrame as a Delta table. We then saw various scenarios, like simultaneous read and write from the table, failure in writing, and updating the records in the table. And all this is possible because of transaction log information stored with each file. And finally, we ended it by learning about additional features and the use cases. This brings us to the end of this course where we started by learning about Azure Databricks, its background, features, components, and architecture. We then saw how to set up the environment, and then we worked on the extract, transform, and load sets of our pipeline. We then made it production-ready and saw how to schedule that using jobs or Azure Data Factory. And finally, here we learned about components that can help us build better ETL pipelines. I hope you had a good experience learning this great service. Thanks for watching, and continue learning.