Combining and Shaping Data

by Janani Ravi

Start Course

Bookmark Add to Channel Download Course

Table of contents Description **Transcript** Exercise files Discussion Learnin

Course Overview

Course Overview

(Music playing) Hi, my name is Janani Ravi, and welcome to this course on Combining and Shaping Data. A little about myself, I have a master's degree in electrical engineering from Stanford and have worked at companies such as Microsoft, Google, and Flipkart. At Google I was one of the first engineers working on real-time collaborative editing in Google Docs, and I hold four patents for its underlying technologies. I currently work on my own startup, Loonycorn, a studio for high quality video content. Connecting the dots is becoming the most sought after skill these days for everyone ranging from business professionals to data scientists. In this course, you will gain the ability to connect the dots by pulling together data from disparate sources and shaping it so that extracting connections and relationships become relatively easy. First, you will learn how the most common constructs in shaping and combining data stay the same across spreadsheets, programming languages, and databases. Next, you will discover how to use joins and VLOOKUPS to obtain wide datasets and then use pivots to shape that into long form. You will then see how both long and wide data can be aggregated to obtain higher level insights. You will work with Excel spreadsheets, SQL, as well as Python. Finally, you will round out the course by integrating data from a variety of sources and working with streaming data, which helps your enterprise gain real-time insights into the world around you. When you're finished with this course, you will have the skills and knowledge to pull together data from disparate sources, including from streaming sources to construct integrated data models that truly connect the dots.

Exploring Techniques to Combine and Shape Data

Module Overview

Hi, and welcome to this course on Combining and Shaping Data. In this module, we'll focus on exploring different techniques that you can apply to combine and shape your data so that its set up in a form that is useful for data analysis. We'll first start off by discussing the two forms in which data can be expressed, wide data formats and long data formats. Wide data formats have all of the information associated with an entity in a single record. As more attributes are added, the table gets wider as we add more columns. We'll see that long data formats are created representing sparse data, and where we don't know the schema up front. After discussing these data-shaping techniques, we'll move onto ways that we can combine data together. We'll discuss the types and uses of joins. We'll then talk about grouping operations that can bring data together associated with the field of interest. And then we'll talk about how we can apply aggregations to our data once data has been grouped together.

Prerequisites and Course Outline

Before we dive into the actual course content, let's talk about the prereqs that you need to have in order to make the most of your learning. This course uses a number of different technologies, and you need to have a basic understanding of working with Excel spreadsheets. You'll need to know a little bit of Python programming to perform some of the demos, and you'll also need some basic SQL to work with relational databases and data warehouses. The only math that you need for this course is high school math. Please note that you don't need to be an expert here in either Excel, Python, or SQL, just basic knowledge is sufficient. Let's take a look at what we'll cover in this course. We'll start off by discussing basic principles in combining and shaping data. We'll talk about wide form, long form data, and combining techniques such as aggregations, joins, and unions. We'll then move onto actual implementations, and we'll perform hands-on demos using spreadsheets, relational databases, and Python. We'll then see how we can bring together data from disparate sources into a single data warehouse for analysis. For this we'll use ETL pipelines, or extract transform load pipelines. And finally, we'll take a quick look at how you can work with streaming data using a data warehouse. All of this on the Microsoft Azure platform.

Connecting the Dots: Combining and Shaping Data

Some powerful person once said, "My mind is made up. Don't confuse me with facts. " Well, I'm glad everyone doesn't feel this way because then this exciting field of data analysis and science wouldn't have come up the way it did. As someone who works with data, you'll often be asked for your thoughtful, fact-based point of view on some topic. If you want to have all of the right facts with you, it has to be built with painstakingly collected data. Your opinions need to be thoughtful. They have to be balanced and weigh the pros and cons of each decision. Analysis that present just one view lose out on the bigger picture and are frankly not that good. And finally, point of view. As an analyst, you might be asked for a prediction, recommendation, or a call to action, and these will be key components that drive the business. As a data professional, there are two sets of statistical tools that are available to you and that you'll use often. Descriptive statistics are helpful in identifying important elements in a dataset. These statistics give you quantitative and summary information about the set of data that you're working with. It tells you nothing about the population as a whole. And that's where inferential statistics come in. Inferential statistics help explain the important elements that you've identified via relationships with other elements, allowing you to infer and deduce information about the population as a whole from which the data is derived. Two sets of tools, two hats of a data professional. As a data professional, you're responsible for finding the dots, identifying the important elements in a dataset. You're also responsible for connecting those dots, explaining those elements via relationships with other elements so that data is actually useful. Finding the dots may seem easy in this world of plentiful data, but that's not really the case. There is a problem of too much, and, of course, this too much data may not always be in the right form for analysis. Careful handling is needed before data can be useful. Before you use your data for analysis, you need to explore it to make sure there are no missing values or outliers. Outliers could be genuine outliers or points that have been erroneously measured. This is just the data cleaning operation. Data then has to be prepared so that it's ready to be analyzed. There are a variety of tools available to you to connect those dots. You might use spreadsheet programs, such as Excel. You might use high-level programming languages that allow you in-memory processing or even distributed processing on a cluster of machines. That's needed when you have data that's very large. If coding is not your thing, you might want to work with data using SQL, or structured query languages. In such cases, it's useful to have data stored in relational databases or data warehouses. Now that we've understood the steps involved in the life of a data professional, when we analyze this data, let's talk about combining and shaping data. Doing this smartly makes it easy for a data scientist to connect the dots. Think of these as two broad categories of operations that you might perform to get your data into a useful format. Shaping data might involve the use of wide forms and long forms to express your data. Certain kinds of data, typically data that is dense in nature, is better analyzed using wide forms. Other kinds of data, that is, data in the real world that is sparse, might require the long form. In the real world, data is typically available in disparate sources, and you need to bring them together in a meaningful way. Some of the techniques that you might use to achieve this are joins, aggregates, and unions. All of these operations can be performed on a variety of different technology, and you'll choose the right one based on your needs. If you want fast prototyping, Microsoft Excel spreadsheets are great, but these are typically brittle and bad for production use. Your code can be more robust and flexible if you use a high-level programming language, such as Python. This gives you fast prototyping in a read-evaluate-print loop environment, that is an interactive environment. The data that you work with here can't be huge because we're still working on a single machine. If SQL is your thing and not coding, then you might go with the Azure SQL database. These are for business users who can't code, but not yet big data. There still exists the problem of silos. If you want to program, your data is huge, and you want access to a distributed cluster, Python with Spark gives you fast prototyping with big data. If you want to work with SQL for analytical processing on huge datasets, you might want to use a data warehouse. The Azure SQL data warehouse is the right choice for you.

Wide Form and Long Form Data

In this clip, we'll discuss wide form and long form data, understand what exactly they mean, and when we would choose to use either one of these. Under shaping data, you can work with data in wide form, as well as long form. And you might reshape the data to meet your needs. If you are wrangling data, you're probably working with data in a tabular format. The traditional wide form has a number of columns and as more data is added, more columns are added to your table and your table gets wider. Imagine that this table here contains information for customers of an e-commerce site. Every customer is identified using the ID and all attributes which are associated with a customer are in the form of columns. This is the traditional wide form. In order to access a particular attribute of a customer, maybe the type of offer that he received, you'll need to be indexing. You'll need to specify the customer ID, as well as the column that you're interested in.

This is an extremely compact and efficient representation for your data and useful as well if your data is dense, if you have values for all of the columns for every customer filled in. But if your data is sparse? Well, that's when you would use long form to represent your data. Here is our original data in the wide format. This is how you can convert it to have it represented in the long format. Observe that the table used to represent the same data has become narrower with fewer columns, but longer, more rows. All of the data associated with a single customer is not present in a single row, but it's spread out across multiple rows. As we get additional details for any customer, more rows can be added with the same ID. This makes long form schema very flexible indeed. Information that was previously stored in columns are now values in a row. The three columns in our original data are now cell values in one of the columns. As we have more properties and corresponding values for a customer, you'll have multiple rows per record and you'll add more rows. And this flexible schema is what makes long form very efficient in representing sparse data. Missing values are not stored in the table. Wide form data requires that you have a slot allotted to each value whether or not it's present. Not so with long form data. If you happen to have data that's not very large, it's well-defined and dense, that's when you choose to represent it in the traditional wide form. But in the real world, data tends to be sparse and requires a flexible schema. Not all of the information is available up front, and that's when long form is preferred.

Joins

It's often the case that all of the data associated with a particular entity is not present in a single table, and that's when we need to perform joins to combine data. Let's take a look at what exactly it means to join data that is present in two tables. Here are two tables, the first of which contains name and salary information for an individual, and the second contains name and the department where that individual works. In order for join operations to be performed between these two tables, you need to have a column that is common between the two so that the join is performed on this common column. Joins are often represented by a cross, or an X. The objective of a join is to have records from each relation or table match on the join column so that we have a single row which contains all of the details for a single entity. What we just performed here is an inner join typically referred to as just a join. This is where records will be present in the result on when there are matching records in the other table. Records in either table that do not have a match in the other table are simply eliminated from the result. When you perform joins using an exact match on the joined column, these are referred to as equi-joins, and these are the most common type of joins. For the rest of this clip, we'll focus our attention on discussing equi-joins. Joins on a whole can be divided into five broad categories, the left outer join, the right outer join, the full outer join, the self join, and the cross join. Let's first consider the left outer join and see what it means. Now when you're performing a join between two tables, you have a table on the left and you have a table on the right. Both of these tables refer to records for individuals, but on the left, you have records for Tom and John which are present in the right table as well, but Judy is missing in the table on the right and Emily is missing on the table on the left. The result of the left outer join will preserve every record on the left table. So every record in the left table will be presenting the result either with a matching record from the right table or the values from the right table will just be null. And here is what the left outer join of the two tables that we saw before looks like. Observe that Judy doesn't have a corresponding match in the right table, so department information is simply null. Let's now consider the same two tables that we were working with to understand the right outer join. The table that is on the right here is the department table. A record for Emily is present in the right table, but not in the left. When you use the right outer join, every record in the right table will be present in the result of your join, either with a matching record from the left table or padded with nulls. You'll use the right outer join when the table on the right is what you'd consider your master table, and you definitely want all of the information from the table on your right to be present in the result. And here is what a right outer join of our two tables look like. Observe that the record for Emily has a null in the salary field. There was no salary information available for her in the left table. If the contents of both tables are equally important and you want information from both to be present in the result, you'll go with the full outer join. When you perform a full outer join, the resulting joined table will have records from both tables in the result, either with a matching record from the other table or padded with nulls. The full outer join of our two tables will have four records in the result. Wherever values are missing, we'll just have null. The self join, as its name implies, is simply when a table is joined with itself. Of course, there will be no missing records or non-matching column values to deal with and the number of records in a self join table is simply the number of records in the original table. And finally, we have the cross join, which does not take into account common values in a column. Here are two original tables. When you perform a cross join, every record in the first table will be matched with every record in the second table, irrespective of the common column. This is true for all records. So, John will match with every record in the second table, and Judy will as well. Performing a cross join on the two original tables will give us a table with nine records, three multiplied by three. Joins can be categorized in a slightly different way as well. You can see that joins are one-to-one, where there's just one record in the first table which matches with one record in the second table. Or, joins can be one-to-many, where one record in the left table matches with several records on the right table. Or joins can be many-to-many, where several records in the first table match several records in the second table.

Aggregations

Data in the raw format doesn't give us much information. One way to make data useful is to combine it using some kind of aggregation operation. You can perform aggregations on the dataset as a whole, a total count of all records. The total sum of all individuals living in a neighborhood. But aggregations are more useful when you group on a particular column. Every row here is an individual record corresponding to an order placed on a e-commerce site. These are records made up of fields. Observe that all of these orders are grouped together using an ID. There are several orders using the same order ID. In order to get information on a per-order basis, the first step that you need to perform is this grouping operation. If you group by order ID for each order, you'll have a set of records. Here are all of the records in o1, o2, and o3. This is the first essential step is any aggregation. When you group orders by ID, this creates a grouping where the key used to group these records is the orders field. And the value here is all records that are associated with the same key, all orders that belong to the same order ID. Once you have all of your information grouped by your field of interest, you can then apply aggregation operations. Aggregations are simply functions that can be applied to field values from multiple records when all of those records belong to a group. Now that you have your data grouped by order ID, you might want to perform a count aggregation, where you count the number of different products that are present in each order. The count operation will give you an idea of the largest order or the smallest order. Or a more useful aggregation that you could perform is sum the total amount spent per order. Or you might want to average the total amount spent per order. This will give you an idea of the average order size and allow you to plan accordingly. Sum, average, count, min, max, all of these are common aggregations supported by high-level programming languages, Excel spreadsheets, and even SQL. Pivot operations are closely related to wide form and long form data and aggregations that you want to view in your data. Pivot operations convert data from long form to wide form to enable aggregations. Unpivot operations are the opposite of pivot. They convert data originally in the wide form to long form data.

Summary

If you have the same data available in different sources, a way to bring all of this data together is to use union operations. Unions are simple and intuitive to understand, and, in fact, copying data from one table to another is also considered a union. We have two sources of data, we perform a union operation, and the union simply combines these two sources together. And with this, we come to the very end of this module where we discussed different techniques that you can use to combine and shape data. We first understood what it is to have data in the wide data format, and we also saw how the wide format data can be converted to the long data format for more efficient representation. We saw that long form representations of data work well when your dataset is sparse in nature and you require your schema to be flexible. We then studied different techniques that can be used to combine data, starting with joins. Joins bring together different bits of information about the same entity, and there are different types and different uses for these joins. Data viewed in the raw format is often not that useful. In order to perform aggregations, we first need to apply grouping operations on our data to group by meaningful values. Once our data has been grouped, aggregation operations such as sum, count, average, min, max, all of these can be applied on our group data. In the next module, we'll work with Excel spreadsheets to apply the techniques that we just studied.

Combining and Shaping Data Using Spreadsheets

Module Overview

Hi, and welcome to this demo on Combining and Shaping Data Using Spreadsheets. And in this module, we'll work with the most popular spreadsheet program of all time, Microsoft Excel. We'll load our data typically in the CSV format into Excel tables, and we'll see how we can perform join operations across tables using the vlookup function, index, and match functions as well. Vlookup, index, and match might be a little hard to use, so Excel offers this Power Query editor, which makes it very easy for you to join tables, and this is what we'll use next. We'll work with data both in the wide, as well as the long form, and we'll see how we can use Excel to convert wide form data to long form and vice versa. We'll also see how we can perform aggregation operations on our data using the pivot and unpivot functionality in Excel. Pivot, if you remember, is what you use to convert long form data to wide form. And finally, we'll see how we can partition data that is originally in the long form to perform operations such as cumulative sum and rank.

Load Data and Join Tables

In this demo, we'll see how we can work with Excel. We'll load data into Excel files and perform some join operations. Here we are on a new sheet of a brand-new Excel workbook. The name of this workbook is Playstore App Stats.xlsx. The Playstore data that we are going to be working with is a modified form of a dataset that's freely available on Kaggle. I'm going to load data I'm going to work with on this Excel file. Go to the Data option on the menu. This will bring up a ribbon option called Get Data. Click on Get Data and let's access data which is stored in the form of a CSV file, From File, From Text/CSV. This brings up an Explorer window on your local computer. Playstore_apps_base_stats.csv is the file that I want to import into this spreadsheet. This option here will import this data in the form of a table on your Excel spreadsheet. Excel gives you a little preview of this data and also performs some data type detection. I want data types for this data to be based on the entire dataset. That's the option that I have chosen here. Once you confirm that this is the data that you are looking for, go to this Load drop-down here at the bottom and select the Load option and load this data in in the form of a table in your Excel file. Observe that this data has been loaded into a new sheet in your Excel Workbook. This sheet is called Sheet2. Sheet2 contains the Playstore-based statistics. I'm going to load in some more data into this Excel Workbook. Go to the Data option and select Get Data. Once again, I'm going to load in data from a CSV file. This CSV file is also present on my local machine. This is the playstore_apps_adv_stats.csv. Import this data in, switch the drop-down to perform data type detection based on the entire dataset. You can see that this dataset also contains Playstore app details. It contains a few additional details, advanced statistics as compared to the previous data that we loaded in. Click on the Load option and load this data in into a separate sheet, Sheet3 of your Excel Workbook. You can see that the app column is common between the two sheets. These sheets contain details about Playstore apps, but you can see that the other columns are different. Here on the sheet we have category, reviews, install, and a bunch of other information about Playstore apps. Now let's see how we can join these two bits of data together to have one table with all of the information. I'm going to select all of these columns here, category, reviews, installs, type, content, rating, last updated, and current version. Hit Ctrl+C to copy these columns in, switch over to Sheet2, and paste these columns at the very end of this table. We'll see how we can use Excel to perform join operations so that all of the data is available on a single sheet. Join operations in Excel can be performed using the vlookup function. This is the function used to look up and retrieve data from a specific column in a table. Vlookup takes in a number of arguments. The first is the lookup value, then the table where we want the lookup to be performed, then the column index that we want to access from that table, and an optional range lookup. We're going to perform a join operation here by looking up the category of the Playstore app, and we'll perform this lookup by the app name. The app name for this particular row for this particular app is specified in the A2 cell of this sheet. We are currently on Sheet2. The category for the application is present in Sheet3. So we reference the table A2 through B265 present in Sheet3. This is the table that we'll use to retrieve the lookup value. The third argument we pass into vlookup is the column that we want to look up in that particular table. We want to look up the column at index 2. Indexes in Excel start at 1, so index 2 refers to the category column on Sheet3. And finally, the last input argument, FALSE, indicates that we want an exact match for the Playstore app name, not an approximate match. We can't really see column A2 here in this view, so I'm going to scroll over to the left so that you can see the app name that is present in column A2, and this is what we are going to use to look up in the table range specified in Sheet3. The app name can be thought of as the join column. Go ahead and hit Enter, and because this is a table, you'll see that all of the values in the category column will be filled in with the app category. In a table, the formula is automatically applied to all cells in the category column. Now that we've understood the basic structure of the vlookup Excel formula, in order to join information in two tables, we can apply vlookup again to find the reviews information for each Playstore app. This is the same formula that we worked with earlier. The only thing that changes is the range, the table range where we want to perform the lookup. The table range goes from A2 to C265. We're including column C here because that's the column in Sheet3 that contains the review information. And the column index number which we want to use to retrieve the information for reviews is column 3. Hit Enter and the entire cells in this column will be populated using this formula, giving us the reviews information. Let's switch over to Sheet3 and confirm that column C is indeed where review count for each app. I'll now switch back to Sheet2 and continue populating the remaining columns using vlookup. Here I'm going to look up the installs information for the app, which is present in column B of Sheet3. Now that we've understood how vlookup works, we'll use a different Excel function to populate the values for the type column using a join operation. In order to populate the app type information, we'll use a combination of the index and match functions. The index function in Excel is used to look up a particular value in an array or a table. Index takes in the range of the array or the table where we want to perform the lookup, the row number, and, alternatively, the column number if you choose to specify it. The value that we want to look up using this index function is present in Sheet3 column E. This is the type information for a Playstore app. The type of an app indicates whether an app is free or paid. In order to specify the row number that we want to look up using the index function, we use a nested match Excel function. The match function searches for a specified item in a range of cells and returns the cell where a match is found. Match takes as its input the value that we want to match in the array or table where we have to look for this match and a match type. For this particular row, we want to look up the app name that is present in cell A1 and we want to look for this app name in Sheet3 A2 to A265. That is the table range. When you specify 0 for the last argument that is match type, Excel will return the first value that was an exact match. Hit Enter and Excel will populate all of the rows in this particular column with this formula. And using a combination of index and match, the app type information is now available on this sheet. Now in order to populate the other columns here, I'm going to perform the join using the combination of the index, as well as the match functions. Only the input arguments will change. The only change in the index match formula here is the column where I perform the lookup in the index function. In Sheet3 I'm going to look at column F, which contains the content rating information. Let's use this index plus match operation to populate the other columns as well, the last updated column, which is present in Column G of Sheet3. The last updated column is actually date information, but it's available in the form of a number. Let's switch the data type to be Short Date, and you'll find this information represented in the Date format, which is what we want. And, finally, let's perform one last index match operation to populate the current version. I'm going to look up column H in Sheet3, which contains the version information. As usual, we'll hit Enter, and because this is a table, Excel will populate this formula in all of the cells of this column.

Aggregation Operations

Let's now scroll to the very bottom and perform a few aggregation operations in Excel using Excel functions. We'll first calculate the average rating across all apps in the Playstore. Excel offers a very handy average function to calculate the average value, and I'm going to pass in the output of the average function to the round function so that I get the information rounded to two decimal places. This formula calculates the average of Sheet2 Column B1 to B265, which contains the ratings information. You can see that Excel performs this color coded highlight of the range on which you're performing the operation. Hit Enter, and you can see that the average rating across these Playstore apps is 4.31. I'll now calculate the total number of reviews across all of these Playstore apps using the sum operation. The number of reviews per app is present in Column F, and that's what I'm summing up here. Once we have the total number of reviews calculated and stored in cell E268, we can calculate the average number of reviews per Playstore app. In the numerator, we have the total number of reviews present in E268 and in the denominator we count the number of Playstore apps. This gives us the average number of reviews per app, which is roughly 77, 000. If you want to count records in Excel based on a certain condition, you can use the countif Excel function. Let's count the number of free apps by checking where the free keyword appears in Column H. Hit Enter, and in our dataset 262 apps are free. Let's see the number of paid apps by using the countif function as well. In Column H only count those records which have the paid keyword. Hit Enter and you'll see that there are just two paid apps in our dataset.

Join Tables Using Power Query

In this demo, we'll see how we can perform join operations in Excel in a much easier way using the Power Query tool. Here we are in a brand-new Excel Workbook called Car Sales.xlsx. Go to the Data option here on your menu, go to Get Data. We're going to load in data once again, but this time our source data is stored in another Excel Workbook. Choose the From File option, From Workbook, and then select the sales_data workbook that I have on my local machine. Click on the Import button and let's import this data in the form of a table. Select the shop_sales_data sheet within this workbook and click on the Load option. You can see a preview of what we are about to load in here. This is some sales data that I made up for car sales at a particular distribution center. This data loaded into Sheet2 contains cars that were sold, when they were sold, the sales rep that sold it, and a bunch of other information. I'm going to load in more data into this Excel Workbook. Go to the Data option, select Get Data, select that you want to load in data from a file. I'll choose an Excel Workbook once again. In the dialog that pops up, choose the emp_ids.xlsx file. This contains the employee IDs of the various sales representatives. In this preview, you can see that Excel hasn't correctly identified the headers in this data. So I'm going to click the Transform Data button here in order to edit this information. This brings up Excel's Power Query Editor that allows you to perform various operations on your data. Choose the Transform option, and select the option here to use the first row as a header. You can see that the data has been updated, so that a first row in our dataset containing Sales_Rep and Emp_Code is now the header. This data will be loaded in as a table on a new sheet. I'm going to name the table Employee Table. Once you have this data set up the way you like it, go to the File menu option and choose Close and Load. This closes the Power Query Editor and loads the data in onto a new sheet. This is Sheet3. I'm going to load in one more dataset here. Go to the Data option, Get Data, and let's load this data in from a CSV file. This is the prod_id.csv file, which I'm going to import into this Excel Workbook. This contains the IDs for each product. Each kind of car sold by our sales reps has its own product ID. I'm going to transform this data so that headers are correctly identified. Go to the Transform option, select Use First Row as Headers, and choose the sub-option here, Use First Row as Headers, so that our dataset has the right headers, Product and P_code. Let's give this table a name in the Power Query Editor. I'm going to call this the Product Table. Go to File and choose Close and Load. This will load the product ID data into a new sheet that is Sheet4. Now that we have all of the data loaded in, let's switch over to Sheet2 where we had loaded all of the sales data, that is, cars sold by the various sales reps. Join operations can be performed using the Power Query Editor that we use to load in and transform the data. Go to Data, Get Data, and choose the option here to Combine Queries and Merge datasets. This will bring up an editor where you can specify the tables that you want to use to perform your join. The first table here is the table from the current sheet, shop_sales_data. You can use this drop-down here at the bottom to specify the table that you want to join with. I'm going to join the shop_sales_data with the product table. Once you've chosen the second table, you can see a preview here within this window. The next step is to select the columns on which you want to perform the join. I'm going to join these two tables based on the product column. At the bottom here is a very useful little drop-down allowing you to specify the kind of join that you want to perform. The default is a left outer join. I'm going to join these two tables using an inner join. Only matching rows from either dataset will be present in the resulting joined table. Click on OK here to perform this merge and join operation. This will bring up the Power Query Editor where you can edit and transform your data. If you select this little icon next to the Product Table column, this will allow you to select what columns in the Product Table you want to include in this join. I only want the P_code column. Observe that the column now shows the product code from the Product Table. I'm now going to edit the name of this column, select the column header, right-click, and choose the Rename option. I'm going to rename this column to be called Product ID. I want this column to be present next to the product name. I'm going to select this column and drag the header to be right next to the product name to its left. I'll now give this joined table a new name. I'm going to call it Merged_Sales_Data, and once this is done, choose the Close and Load option so that this data is now loaded in as a new sheet in this Excel Workbook. Remember we also have the employee ID and employee name information in this Excel Workbook. You can perform a join operation between the sales data and employee table using the employee name. The join operation can be done using the Power Query Editor, applying the same steps that we used in this demo. I'll leave that for you to do as an exercise.

Convert CSV Data to Table Format

In this demo, we'll see how we can use Excel to convert wide form data into long form. The original wide form data that we'll work with is present in the form of a CSV file. Double-click on this file, and this opens up in the form of an Excel Workbook on my Windows machine. We are working with the honey production across different U.S. states dataset, which is freely available on kaggle.com. This is a CSV file. I'm going to save this as an Excel Workbook so that I can operate on this data. I'll call this workbook Honey Production and instead of a CSV file I'll choose its format to be an Excel Workbook. Click on the Save option and save this file in the Excel format. The best way to explain wide form data is data that is horizontal in nature. Notice that our rows here are the different states and the columns are different years, and for every year we have the honey production by state. I'll now convert this CSV data to a table format in Excel. I'm going to select all of this data and use this plus icon here to create a new sheet. And I'm going to paste in this data onto this new sheet and we'll convert this sheet to be in the table format. Select all of the data on this sheet that is Sheet1 using your keyboard and mouse and go to the Insert menu option. There is an option for Table here, and specify the data range for your table. If you have all of the data in this sheet selected, the data range will already be prefilled within this dialog window. Click on OK and observe that your data has now been converted to a table format. Certain operations, such as join operations, are best performed on tables so it's useful to know how to convert raw CSV data to a table format. I'm now going to visualize some of this data in Excel. I select the first two columns for state and the year 1998. I go to the Insert Option, and I'm going to choose Recommended Charts. You'll see that Excel suggests a bunch of charts that I can use to visualize this data. I'm going to select the Pareto chart here, which plots the distribution of the data in descending order of frequency and accumulative line as well. You can see that Excel inserts the Pareto chart for your selected data right within your sheet. With this chart selected, I'll hit Ctrl+X to cut this chart, scroll down to the bottom, and paste it here at the bottom away from the rest of my data. You can see that Excel makes it very simple to convert your data to a tabular format and visualize it using a variety of charts.

Convert Wide Form Data to Long Form

Here we are in the Honey Production Excel Workbook on the first sheet where we have the honey production details in the wide form. In order to convert this data to the long format, I'm going to pivot this data. You can bring up the dialog for the pivot table by hitting Alt+D+P. This will bring up the wizard that you see here onscreen, the pivot table, and pivot chart wizard. Select the option to create a pivot table using multiple consolidation ranges. This is a pivot table that we'll create off this Honey Production data, which is in the wide format. Click on Next. Select the option here to create a single page field for your pivot table. A page field refers to the data field that the pivot table uses to summarize your data. Click on Next and specify the range that you want to summarize using your pivot table. I select all of the data that is present here on this sheet. Click on Next and select the option to add this pivot table to a new worksheet within this workbook and click on Finish, and you can see that a new sheet, Sheet2, has been created with your pivot table. The pivot table displays a grand total column at the very end summing up the honey production for individual states across all years. Scroll to the bottom here of this grand total column and select the very last cell. Right-click on this cell and you'll find an option to show the details of this pivot table. And when you click Show Details, your data will be converted to the long form. Observe that Excel has created a new sheet, Sheet4, with this long form data. Long form data is vertically oriented. Instead of having individual columns for each year, we now have a single column for the year. And the cells in that column represent the different years. I'm going to select this last column here, page1, we don't need that information. I'm going to right-click and delete this column. You can see that Excel has auto-generated the headers for this long form data, row, column, and value. I'm going to change these headers to be meaningful. They represent the state, the year, and the produced value of honey. We've successfully pivoted our wide form data and then converted this wide form pivot table into the long format using Excel.

Pivot Tables for Summary Statistics

In this demo, we'll see how we can use a pivot table to summarize data in Excel. Here we are on a new Excel workbook called Company Finances.xlsx. And we are on a new sheet called Sheet1. The dataset that we are going to use here to perform pivot operations is this retail dataset available freely on kaggle.com. This dataset includes product information, the segment to which a particular product is sold, the country, the product itself, discounts, units sold, profit, dates, and so on. Let's create a pivot table to summarize this data. Go to the Insert menu on Excel and choose the pivot table option here in this menu. This will bring up a dialog where you can specify the table that you want to pivot. This data is loaded into a table called financials, so I'm going to choose the table named select this data. You can also specify a cell range here. Click on the OK button here and this will create a brand-new pivot table on a new sheet. The pivot table that we just created hasn't been configured yet. We'll use this pane on the right-hand side to configure the pivot table fields. I want to summarize data based on the products sold. I'm going to select the product field from my original table and drag it down to the rows section. Observe that our pivot table has been updated to have the product as the row label. A pivot table will summarize information on a per product basis. I want to be able to filter my pivot table information by country, so I'm going to select the country field and drag it into the filter section. You can see that country has been added as a filter at the very top of your sheet. Let's now select the fields that we want to summarize. I'm going to select the Units Sold field and drag it into the summarization of values section. The default summarization for an Excel pivot table giving me the sum of units s sold per product. I'll now summarize the information in a few more fields, select the sales price field, and drag it onto the summarization of values, giving me the sum of sale price of each product. One last field that I'm going to include here is the Profit field. Select the Profit field and drag it into summarization of values, giving me by default the sum of profit that the company made for each product. Here is our fully set up pivot table. Now, the font here is rather small, so I'm going to select the pivot table and change the font size to be 16. Now that we have our pivot table all configured, I'm going to edit the summarization columns for our pivot table. Double-click on the Sum of Profit column and this will bring up an Editor dialog. The default aggregation that an Excel pivot table applies is the sum aggregation. You can select another aggregation if you wish to. I'm simply going to change the name of this particular column to be Total Profit so that it's meaningful. Click on OK, and you'll find that our summarization column header has been updated. Double-click on the column for the Sum of Sale Price, bring up the editor dialog. This time I'm going to change the aggregation that I perform to find the average sales price per product. Update the column header as well to say Average Sale Price. Click on OK. And this column in the pivot table has also been updated. Let's change the column header for the Sum of Units Sold. Double-click on this column header, bring up the edit dialog, and call the column name Total Units Sold. The pivot table in Excel is extremely useful, allowing you to quickly summarize data based on a category. Here the category is products. I'm going to update the row label so that the column header is meaningful. I'll set it to products. Observe that the pivot table gives you a little drop-down here for a category, allowing you to select the categories that you want to view. By default, all categories present in the original table have been selected. I'm going to uncheck Select All and then select specific category. I'm going to select three separate categories. Click on OK here, and you'll see that the pivot table now displays just three categories. We've also configured an additional country filter for our pivot table. You can click on this little drop-down icon. This will bring up an editor window, allowing you to further filter your data by country. I've only selected Canada here. My pivot table now displays summary statistics for the three products that I selected for the country Canada. Both of the filters have been applied on my pivot. If you want to see how these products perform for a different country, select the country filter once again. Let's change the country selection to United States, click on OK, and you'll see all of the products' summary statistics for the United States of America. Thus, pivot tables allow you to drill down into your data in a very granular form. Let's select the products filter once again, select all of the products, and this view will show you how all of the products offered by your company performed in the United States. This view tells you that in the United States, the most profitable product sold by our company is the paseo.

Unpivot Tables

In this demo, you'll see how we can unpivot wide form data into long form and keep the original pivoted data and the unpivoted data in sync. Here we are in a new Excel workbook. Sales Rep Numbers is the name of this workbook. I'm going to load some data into this Excel workbook. This data is originally in the wide form. Choose Get Data, From File, From Workbook, and select the monthly sales Excel workbook to load in data. Click on Import, and this will bring up the Import Editor. There is a table within this workbook called Table1. That's what I select here. And I'm going to transform this data that is originally in the wide format to a long format. Observe that this data is made up data, which has the names of different sales representatives and the sales that they made on different months of the year. You can see that the data is horizontally oriented with every month as a separate column. This is wide form data. Click on Transform Data, and this will take you to the Power Query Editor. In order to unpivot this data to long form, click on the Sales Rep column. Right-click and choose Unpivot Other Columns. All of the months that were originally present as separate columns in the wide form data now are in the long form. The months are now cell values in the Attribute column. I'm going to rename this Attribute column to be more meaningful. I'm simply going to call it Month. The units sold by every sales rep for each month is also present as cell values in another column. I've renamed this column as units sold to correctly represent this data. Let's now give this long form table a name. I'm going to call it Unpivot Table. The additional data was pivoted. This is the unpivoted representation. Select the Close and Load option in order to load this data into our sheet. And here on the sheet the original data that was in the pivoted form is now available in the unpivoted long form. Every row of this table represents the units sold by a particular sales rep for a particular month. When you load in data from another Excel workbook like we did right now, Excel keeps these two workbooks in sync. Now let's switch over to the monthly_sales.xlsx workbook, which has this original data in the wide pivoted format. This is the Excel workbook from which we loaded in the original data. This is the data in pivoted form. Now I'm going to make a little change to this original notebook. I'm going to copy over this last row and paste it in to include a new sales representative. I'm going to call the sales representative Harvey and I'm going to change some of his units sold numbers. What I've done here is change the data in the original workbook, which we use to load in data to our workbook that we created, the sales rep numbers workbook. Hit Ctrl+S to ensure that this new update has been saved on this monthly sales workbook. Now let's switch over to Sales Rep Numbers. This is the Excel workbook with the data in the unpivoted long format. Observe that we have a panel here on the right indicating queries and connections. This is where we know that this workbook is connected to the original workbook from where we loaded in data. Click on this icon here on top. This represents the query which we used to unpivot our data. This refreshes the data in this workbook so that the new information for Harvey is now included here. Saved changes in the original workbook are automatically reflected in a connected workbook.

Calculating Cumulative Sum and Ranks on Partitioned Data

In this demo, we'll see how we can partition data in Excel to perform operations such as cumulative sum and rank. Here we are on a new Excel workbook called Coffee Sales Metrics.xlsx. Here we have a dataset which I've just made up here containing information about store sales. All of these coffee stores. The columns here represent the name of the store, the date on which the sales were made, and the total sales. And this information is available for three different coffee stores, The Split Bean, Manhattan Mocha, and Camdens Coffee. If you scroll down, you'll see that the data is not really sorted. It's all jumbled up and all presented together. Now before we can partition and analyze this data, it needs to be cleaned up a little bit. Select all of the rows and columns of this dataset, select the Sort and Filter option under Home, and choose Custom Sort. This brings up a dialog where you can specify sorting options by multiple columns. I first want to sort by Store. So I want all of the store's data for a single store to be located together. I'm now going to add another level into my sort and I'm going to sort by date for each store. All of the records that I've selected in my dataset will first be sorted by store and within each store it'll be sorted by date. This custom sort will allow us to partition our data by store. Click on OK and here you can see that the data is cleaned up and ready to be analyzed. We first have all of the sales information for Camdens Coffee, and you can scroll down and get information for Manhattan Mocha and The Split Bean as well. I first want to calculate the total sales on a per-store basis. I'm going to select all of my dataset, choose the Data option, and within here choose the option which says Outline. The Outline option gives me different ways to summarize my data. Here is where I can find an option to calculate subtotals. Click on subtotals. I want to calculate the total sales for each store. Once you hit OK, you'll find that all of your data will be grouped by store and you'll get a subtotal for each store. Here is the total sales for Camdens Coffee, and if you scroll down below, you'll find that a row has been inserted giving you the total sales for Manhattan Mocha, and if you scroll further down at the very end, you'll find the total sales for Split Bean, as well as a grand total. The grand total gives you the total sales across all three coffee shops. Let's go back to the top here. We've successfully partitioned the data by store and calculated subtotals. I'm now going to right-click and insert a new column. And here in this column I want to keep a tab on cumulative sales, cumulative totals across each day for each store. For the very first day for Camdens Coffee, the cumulative total is simply the sales that were made on that particular day. So we say equal to C2 in the cumulative sales column. For the first cell here, sales is equal to cumulative sales. Let's set up the formula for the cumulative sales in the second cell. I've used the Excel formula to sum up all of the sales starting from the first day, that is, at C2, up to the current cell, C3. I've used C2 so that this cell is locked. When we copy this formula down to other cells, C2 will remain unchanged. Hit Enter and this will give you the cumulative sum for the first two days. I'm now going to select and drag this cumulative sum formula that I've populated on this cell down to all of the cells for Camden Coffee. Now within this column, I'll have the cumulative total for each day for which I have sales data. So for day 3 here within Camdens Coffee, the cumulative total reflects the sum of sales for the first 3 days. For day 4 it reflects the sum of sales for the first 4 days. In exactly the same way I can calculate cumulative totals for the other coffee shops as well. In the first cell for the Manhattan Mocha coffee shop, the cumulative total is simply the sales for that particular day equal to C43. Hit Enter to populate this value in this cell. In the second cell, I'll set up the formula for cumulative total. The cumulative total is the sum of all sales data starting from cell C43 that is locked up to the sales cell in the current row, that is, C44. Hit Enter and you'll get the value for the cumulative total for the first two days. Select and drag this formula down to populate cumulative totals for the sales data for the Manhattan Mocha coffee shop. Hit Enter and all of the cumulative totals will be populated. I'll perform the same set of operations to calculate cumulative totals for The Split Bean as well, equal to C84, then set up the formula for cumulative totals, and once the formula has been populated, drag it down to fill all of the other cumulative totals. Now that we've calculated cumulative totals for our partition data, I'm going to move off to the side and see the rank of the various stores by revenue. I'll first populate two columns, the name of the store and the total revenue for that particular store. The three stores are Camdens Coffee, Manhattan Mocha, and The Split Bean. The total revenue for each of these stores have already been calculated using our subtotal operation. Let's now find the rank of each of these stores. We can use the RANK function in Excel for this. The RANK function takes in three bits of information. The cell containing the data that you want to rank, then the reference cells amongst which you want to find the ranks, and finally, the order, whether you want the data to be ranked in ascending or descending order. Cell C42 here contains the total revenue for Camdens Coffee, and I want to find its rank amongst the three cells that contain the total revenue for Camdens Coffee, Manhattan Mocha, and The Split Bean. These are present in cells C42, C83, and C124. The 0 here indicates that I want the data to be ranked in descending order, that is, the highest value has rank 1. If you hit Enter, you'll see that by revenue, Camdens Coffee is at rank 2. We'll use the same rank formula to calculate Manhattan Mocha's rank by revenue. The only difference here is what we want to rank, that is, the revenue of Manhattan Mocha is present in cell C83. When you hit Enter, you can see that Manhattan Mocha is at rank 3 based on revenue.

I'll now apply the rank formula once again to the revenue number for The Split Bean, which is in C124. Hit Enter and you can see that The Split Bean is at rank 1.

Summary

And with this last demo on partitioning data to calculate cumulative sum and rank, we come to the very end of this module. We started this module off by understanding how we could perform join operations in Excel using tables. Excel offers some very powerful functions that you can use. We performed join operations using vlookup and the index and match functions. Excel makes common operations easy. We then saw how we could use the Power Query Editor to join two tables together using a common column. We saw that the Power Query Editor was simple and easy to use and it eliminated the need for writing complex functions that we might get wrong. We work with data in Excel that was both in the wide form, as well as the long form. We saw how we could use Excel functions to convert wide form data to long form and vice versa. We then performed the pivot and unpivot operations on Excel tables, and we also saw how we could set up linked workbooks. Changes that we made to the data in the source workbook were reflected in the workbook that referenced the source. And finally, we saw how we could partition data expressed in the long format to perform operations such as cumulative sum and rank. In the next module, we'll move on from Excel spreadsheets to writing queries in SQL. We'll work with the Azure SQL database on the Microsoft Azure Cloud platform.

Combining and Shaping Data Using SQL

Module Overview

Hi, and welcome to this module on Combining and Shaping Data Using SQL. If high-level programming languages are not your cup of tea, but you're proficient with the structured query language or SQL queries, you can use SQL queries to combine and shape your data. In this module, we'll run SQL queries using the Azure SQL database. This is the database service offered on the Microsoft Azure platform, and there are several deployment options that you can choose from. The Azure SQL database allows you to run queries using a browser-based query editor. There's nothing additional that you need to install. We'll load data into our cloud-based SQL database using the Azure Data Factory. These allow us to set up extract, transform, and load pipelines to copy data to the Azure cloud from disparate sources. Once data is available within our database, we'll see some example aggregation operations that we can perform in SQL. We'll set up more than one table and perform different kinds of join operations, the inner join, the left join, the right join. And, finally, we'll also study pivot and unpivot operations in SQL to take a more granular look at our data.

Introducing the Azure SQL Database

In this module, we'll work extensively with the Azure SQL database on Microsoft's Azure cloud platform. Hopefully you're already familiar with the idea of SQL and relational databases. The Azure SQL database is the managed relational database service offered by Microsoft, and it's ideal for transaction processing applications. This is Microsoft's platform-as-a-service offering on the Azure cloud, and it shares its code base with Microsoft SQL Server database engine, and you'll find that working with SQL Server on-premises is very similar to working with the Azure SQL database on the cloud. This is a powerful and useful technology because Microsoft is now cloud-first. Any new updates to the database engine goes to the Azure SQL database first. It's a fully managed service, which means that administrating this database is very straightforward. There is no overhead for patching or upgrading the database. When you're working with SQL databases on the Azure cloud, there are three deployment options that you can choose from. You can set your database up as a single database, which has its own set of resources managed via a SQL database server. A single database is isolated from other databases and from the instance of SQL server that host the database. Or if your compute and data needs are not very high, you might choose to set up your Azure SQL database as an elastic pool, which is a collection of databases with a shared set of resources. The third deployment option that you'll have is a managed instance, which is a collection of system and user databases with a shared set of resources. In the demos that we'll work with, we'll set up a single database on the Azure cloud. We'll use the Azure Data Factory to load data into our SQL database. This is a managed service meant for building complex, hybrid ETL pipelines that integrate data silos, and these pipelines can include Hadoop and Machine Learning transformations. ETL here stands for Extract, Transform, and Load operations that you perform on data.

Loading Data to Azure Blob Storage

In this demo, we'll work with the SQL database created on the Azure cloud platform. We'll see how we can load CSV data that is present on our local machine into an Azure SQL database on the cloud using the Azure Data Factory. I'm going to load in three separate files into the Azure SQL database. The first is this hostel dataset that's present in the hostel_dataset.csv file. This is a subset of the original hostel dataset that's freely available on kaggle.com. This dataset contains lodging data from hostels around the world. The hostel ID, the name, city, starting price, distance from the city center, and the average rating. Another subset of the same data is present in a different CSV file, hostel_booking.csv. This file contains booking information for hostels, the booking ID, the hostel ID, and the number of rooms booked. And the third dataset that we'll use in this module is the honey production dataset. This is one that we are familiar with. And you can see that it contains honey production information in the long form, state, year, and produce value. In order to work on the demos of this module, you'll need a subscription to the Azure cloud platform. Log into your Azure account at portal.azure .com. I already have an Azure account created and set up. This is a personal account, so that's the choice I make here. I specify the password for this account and then sign in. This will take me to the Azure home page. I'm first going to upload the CSV data that I have on my local machine to a blob storage container on the Azure cloud. For this I need to create a storage account. Click on Storage accounts and click on the Add button here in order to create a new account. All of the resources that you create on the Azure cloud have to belong to a resource group. A resource group is a logical grouping of resources. I'm going to create a new resource group called loony-csd-rg. This is the resource group I'm going to be using throughout this course. Click on OK and specify a name for the storage account that you want to create, loonycsdstorage is my storage account. You can click on the Next button to see what other configuration options you have. I'm going to accept the default values for all of these configuration settings. I'm not associating any tags with this account. Tags are just key value pairs that allow you to identify and group resources. Click on the Review + create button. This gives you one last chance to review the settings for your storage account. Click on Create, and we'll just have to wait a few seconds for this storage account to be created. Clicking on the Go to resource button will take you directly to the storage account that you just set up. There are different kinds of storage resources that you can provision within this storage account. I'm going to create a blob storage container. Click on Blobs and click on the + container button here in order to create a new blob storage container. Blob storage on the Azure cloud is the equivalent of S3 buckets, or Google cloud storage buckets. This container is called loony-csd-blob-container. Click through to this container and let's upload the CSV files that we have on our local machine onto this blob storage. Click on upload and use this browse option in order to bring up a dialog, which will allow you to browse and select files from your local machine to upload to the Azure cloud storage blob container. It should just take a few seconds for all of your data to be uploaded and available on the Azure cloud. Now that this data is available on the Azure cloud, we can run an ETL pipeline to load this data into an Azure SQL database.

Creating an Azure SQL Database and Tables

We'll now create a SQL database on the Azure cloud platform to work with our data using the structure query language. Click on SQL databases using the navigation pane on the left. SQL database on Microsoft Azure is a general purpose relational database managed service. We are going to create a single database which has its own set of resources managed via a SQL database server. We'll place this single database in the resource group that we had created earlier, loony-csd-rg. Give this database a name that you can use to identify it. Loony-csd-sqldb is the name that I've chosen. You now need to create a database server. Click on Create new. Here, because we don't have an existing database server, I'm going to create the loony-csd-sqlserver, and I'll use this dialog to specify a SQL-authenticated login user, loony-csd-admin. Go ahead and specify a password for this user as well. I'm going to locate the server that holds my SQL database in the West US 2 region. Your database server can be uniquely identified by using the server name that you specified, loony-csd- sqlserver.database .windows .net. Select the server and you can choose to configure additional settings if you want to on the next screen. I'm going to use all of the default settings here. Click on Next and add in any tags that you wish to identify this resource. I'll leave this empty. Click on Review + Create. If you wish to review any of the details on the SQL database that we are about to create, this is the page to do it. Click on the Create button here and wait for this resource to be deployed. Once the deployment is complete, Azure will pop up a go-to resource button, which can use to directly head over to our SQL database. I'm going to go ahead and minimize this left navigation pane so that I have more room to work with on this page. Let's select the query editor that allows us to query our SQL database right within our browser. Before you can use the query editor, you'll need to log in using your SQL server username and password. Active Directory Single Sign-on is also available, but we don't have it set up for our account. We're logged into the query editor for the database that we just created. The object explorer pane on the left gives us the tables, views, and stored procedures available. I'm going to minimize all of the panes available to my left so that I have more room for my query editor, and I'm going to write a create table query to create a new table within my SQL database. This table is called hostelData with the hostelID as the key. This is the table where we load in the contents of the hostelData.csv file that we had looked at earlier. Go ahead and click on the Run button here to execute this create table query. At the bottom, you'll get a message saying

Query succeeded. The table was successfully created. Let's now create another table for bookingData. The primary key here is the BookingID for a hostel booking. Click on Run once again to execute this query and create the bookingData table on our SQL database. And finally, let's create the third table that we'll work with. This is the honey production table which holds the state, year, and the produce value for honey. Executing this query will create this table, and you can view all of the tables that you just created by expanding the tables node on this object explorer. Here are the bookingData, HoneyProduction, and hostelData tables.

Loading Data Using Azure Data Factory Pipelines

Now that we have our SQL tables created, we can now load data from our blob storage container into these SQL tables. And for this we'll use the Azure Data Factory. Now in the real world, your data might be present in disparate sources. The Azure Data Factory is a cloud-based data integration service that allows you to create workflows to copy and transform your data and move it to the Azure cloud. Here on the Data Factory stage, I'm going to click on the Add button here to create a new data factory. I'll name it the loony-csd-df. I'm going to use an existing resource group in order to create this data factory. This is the loony-csd-rg that we had created earlier. You might want to locate this data factory in the region where your data is located, or you can choose any other location. Click on the Create button here. This data factory that I've created is in the East US region. Once the data factory has been successfully created, you can click on the Refresh button here and you'll find the data factory listed here on the data factories page. Let's click through to this data factory. Within this data factory, I'm going to author a new data transformation pipeline. Click on Author and Monitor. And this will take you to the Azure Data Factory page where you can set up a pipeline to copy data from a source to a destination. We'll have this pipeline copy data from the blob storage container that we had created to our Azure SQL database. Click on Next and this will take us to a page where we can configure the source of our data. Use this page to create a new data source connection. There are multiple choices available here in this pane. Choose the Azure blob storage because that's where our data is located. The source and destination of your data pipelines are connected via a linked service. This is the LoonyBlobStorageLink service that we are in the process of setting up. We'll connect to blob storage using our Azure subscription, which is a pay-as-you-go subscription and the storage account name is the loonycsdstorage that we had set up earlier. Because this is on the Azure cloud, this is all the configuration that you need to specify. Let's check whether our connection has been successful. Click on Test connection and you'll see that the connection was successfully established. Click on Finish to set up this link service. Select the LoonyBlobStorageLink and click on Next where we can configure the container from where we want to read in the source data. The Browse button here will bring up the containers that you have available. We just have the loony-csd-blob-container. Select this, and within this, let's select the first dataset that I want to load into the Azure SQL database. This is the hostel dataset. Clicking on the Next button here will take you to a screen where you can configure the file format settings for your source dataset. Our Azure Data Factory pipeline has correctly identified this as a comma delimited text file where rows are delimited using the line feed. You can change these options if you want to, but these are the right options. We can go ahead and preview the data here at this lower pane. Everything looks good with our source connection here. Let's click on the Next button and let's move onto configuring the destination for our pipeline. Let's create a new connection. We want to load this data into the Azure SQL database that we just created. Click on continue to set up the SQL database link service. I've called this the LoonyAzureSQLDatabase link. A subscription that I'm using is a pay-as-you-go subscription, so make that selection here. And the server name that we are going to connect to is the loony-csd-sqlserver. Azure will automatically populate the names of databases located within this server. Here we'll select the loony-csd-sqldb that we had created earlier. The next step is to authenticate and authorize ourselves to this server by specifying the SQL authenticated username and password we created earlier. Click on Test connection to check whether this connection is successful. This happy green checkmark here tells us that our destination link service has been correctly configured. Click on Finish, select the LoonyAzureSQLDatabaseLink at our destination, and we can move onto table mapping. Within our destination database, we'll load the contents of our hostelData CSV file into the hostelData table. Select the table and hit Next. The next screen allows you to specify what columns from the source data you want to map to the destination table. I want all of the data from the source CSV file to be loaded into the destination table. I simply hit Next and move on. This is the screen where you can configure performance and fault tolerance settings for your pipeline. I'm going to make a slight change here. If there are any rows in my source dataset that are incompatible with my destination table, I'm simply going to skip those rows. The next screen gives you a quick summary of this pipeline that you've just configured. It goes from Azure blob storage to an Azure SQL database. There are all of the settings that you configured for this copy operation. Click on Next and let's move onto deploying this pipeline. This deployment is now complete. You can choose to monitor this pipeline individually if you want to. I'm going to hit Finish here and set up another pipeline to copy data from my blob storage container to my SQL database. Remember we uploaded three separate CSV files to our blob storage container. This is the second CSV file. The source is the LoonyBlobStorageLink. Let's configure the input file settings. Browse to the blob storage container, and within loony-csd-blob-container, select the hostel_booking.csv file. Click on Choose and let's copy over this data. The next screen is where we preview the data and examine the file format settings. Everything looks good here. Let's click on Next to configure the destination. We load this data into the Azure SQL database that we had created using the same link as earlier. The destination table for this data is the bookingData table. Select this table and hit Next. No change to the column mappings. We want all of the source columns to be loaded into our destination table. Hit Next. Let's skip all rows from our source dataset that are incompatible with our destination table. Hit Next once again. Take a look at the summary of the pipeline that we just created. Hit Next to start deploying this pipeline. Click on Finish and let's set up one last pipeline to copy over the remaining honey production dataset. Click on Copy Data to kick start this new pipeline. This is CopyDataFromBlobToAzureSQLDatabase3. The source is the LoonyBlobStorage. Browse within our blob storage container until you find the honey_production.csv file that we had uploaded earlier. Choose this as the source dataset. Click on Next to preview the file format settings and the data that we are about to load in. Hit Next in order to configure the destination. The destination is once again the Azure SQL database. The table that we want to load this data into is the HoneyProduction table that we had created earlier. The next screen is to specify what columns you want to map to the destination database. Once again, we'll select all of the source columns. We can move onto configuring our pipeline. I want to skip incompatible rows in my source dataset. Hit Next. Here is the page to review the pipeline that you've just created. Click on Next and wait for this pipeline to be successfully deployed. We have now created and deployed three separate copy data pipelines. Let's monitor all three of these pipelines. Click on the Monitor button here. There's only one pipeline visible here. I'm going to hit Refresh to see if other pipelines are also available. I actually have a filter turned on. Click on this filter option. The filter has been set to the last pipeline that we just deployed. I'm going to remove this filter and all of the pipelines that we created are now visible here. All of the pipelines were successful. It seems like our data transfer operations were completed successfully. I'm now going to close this Azure Data Factory tab and head over to the Azure SQL database in order to confirm that our data was successfully loaded. Click on the SQL Databases option. Click on the loony-csd-sqldb, and let's use the query editor to query data. The query editor requires that you login with your SQL authenticated user. Click on OK, and let's run a simple SELECT star from bookingData. Click on the Run button to execute your query, and there you see it. Data has been successfully loaded from our CSV file to the bookingData table using the Azure Data Factory pipeline. I'll now run a SQL query on the hostelData table. A simple SELECT * is sufficient to check whether data has been loaded. And yes, indeed, it has. Now for our last table here, SELECT * from dbo.HoneyProduction. Execute this query and you'll see that data has loaded successfully here as well.

Querying Data in SQL Using Aggregations

With our data successfully loaded onto Azure SQL database on the Azure cloud platform, in this demo we'll perform some SQL aggregation operations to extract insights from our data. We'll write our queries using the browser based query editor into which we are logged in and authenticated. Let's do a simple SELECT * command. When you execute this query, the results will be populated in the lower pane of your browser. If you click on the Messages option, you'll see how many records were returned, 298. The results pane gives you the actual data. The messages pane gives you additional information. From this hostelData, I want to know those hostels where the average rating is greater than 8 for the city of Kyoto in Japan. Executing this query will give you the records for all highly rated hostels in the city of Kyoto. Observe that all of the ratings for these hostels are greater than 8 as specified by our query clause. I'll now run a simple COUNT aggregation query to see how many hostels in the city of Tokyo have an average rating of greater than 8. This count aggregation will give us just a single record in the result. There are 102 hostels in Tokyo with a rating of over 8. I'll now run a SQL query to perform aggregations using a GROUP BY clause. I'm going to select the city and the average of the AvgRating column for all hostels grouped by city. The results will give us an idea the average rating of hostels located in that particular city. The results here show us that on average hostels based in Hiroshima have a higher average rating as compared to those based in Kyoto. Let's run a query to see how expensive hostels are across cities. Select city and the average of starting price grouped by city. If you execute this query and view the results, you'll see that certain cities are comparable to others, such as Hiroshima and Kyoto in terms of starting price, but Osaka and Tokyo are very expensive indeed. I'll now run a SQL query that has a nested subquery. I want to select the hostel, city, distance, and average rating from the hostel data where average rating is equal to the max average rating. I want to display only those hostels which have the highest possible ratings. Let's take a look at the results here after executing this query, and you can see the hostels which have an average rating of 10, which is the highest possible. The count distinct aggregation function will give us a number of unique hostels present in this dataset. If you execute this query, you can see that we have 298 unique records. I'm going to run one last aggregation with the GROUP BY clause. I want to find the minimum starting price for hostels in each city. The results of this query should show you that the minimum starting price for hostels across cities remains more or less the same across all cities.

Performing Join Operations in SQL

In this demo, we'll see how we can perform join operations across different tables using SQL queries. We're already familiar with the data in the hostelData table. I'm now going to perform join operations between the hostelData and the bookingData table. The join column here will be the hostel ID. If you take a look at the Messages tab for this particular SELECT * query, you can see that there are a total of 169 records in the bookingData table. We've already done a SELECT * operation on the hostelData. I'll do so once again, and we know that hostelData contains 298 records. I'll now execute a query that performs an inner join between the hostelData table and the bookingData table. And from the joined tables, I'm going to select the booking ID, the hostel ID, the hostel, and the rooms booked. The structure of the SQL join query remains the same no matter what kind of join it is. Here we want to perform an inner join between the hostelData and the bookingData tables. The join column is specified after the ON keyword, where hostelData.HostelID is equal to bookingData.HostelID. Execute this inner join. The results here will only include those records where there are matching rows in both hostelData, as well as bookingData. If you take a look at the messages, you can see that there are 150 records that are returned. There are only 150 records that match in the two tables. I'll now run another query, this time to perform a left join operation between hostelData and bookingData. The join column is once again the hostel ID. The only thing that changes here is the join keyword. This is a left join. The results here will include all of the records from the left able, that is, the hostelData table, even if there isn't a corresponding match in the bookingData table. Records in the hostelData table which do not have a matching record in the bookingData table will be padded using nulls. I'm going to scroll down and load more of the results so that you can see where this occurs. Observe that here at the bottom there are certain records which have entries in the hostelData table, but there are no corresponding entries in the bookingData table. This is the left join. Let's go up to the query editor and let's switch this left join over to be a right join instead. The right table is the bookingData table, the left table is the hostelData table. When you use the right join operation, the results will include records which have a row in the bookingData table, that is, all of the records from bookingData will be included, even if there isn't a matching record in the hostelData table. BookingData records which do not have a corresponding match in the hostelData records are padded with nulls. Let's look at one last SQL join query here, that is the full join. The full join will include records from both hostelData, as well as bookingData, even if there isn't a corresponding match in the other table. Execute this query and if you scroll down and view the results, you'll find some records that are present only in hostelData. Let's scroll down further and here are other records that are present only in bookingData.

Performing Pivot and Unpivot Operations in SQL

In this demo, we'll see how we can execute pivot and unpivot queries using SQL. The data on which we'll perform these operations is the HoneyProduction data that we loaded into the table HoneyProduction. Let's SELECT * from HoneyProduction, and here are the results. This contains state information, year, and the value of honey produced. You can see that the original representation of this data is in the long form. For every state we have a corresponding year, and for this combination of state and year we have the value of honey produced. Let's now see how we can run a pivot query on this data to see the total value of honey produced per state for a particular year. From the pivoted table, we are selecting the state and the years 2008, 2009, 2010, 2011, and 2012. The table that we are pivoting is the HoneyProduction table, which contains the data in long form. We perform the pivot operation in SQL using the pivot keyword. We pivot to perform a summation operation on the ProduceValue field, sum of produce value for year in 2008 through 2012. Observe that the state information is not part of this pivot, which means that there's the row label. Let's execute this query, and here at the bottom you'll find the pivoted results. The first column here is the state information. There is exactly one record for each state in this pivot table. And the remaining columns in the result are the years that we have selected, 2008 through 2012. You can scroll down to view this pivoted information for all of the states. You'll notice that some pivot values are blanks, indicating that the data is not available. I'm now going to perform another pivot operation, which is very similar to this one, but this time I'm going to store the pivoted data in a new table. Select the state and the years 2008 to 2012 and load this pivoted data into a new table called HoneyProductionByYear. The pivot operation is the same as we saw earlier. The sum of ProduceValue for the year is 2008 through 2012. Execute this query. The results won't be displayed in the lower pane, but the query affected 44 rows that were loaded into the new table that we created. I'll now confirm that this data is indeed present in this new table, SELECT * FROM dbo.HoneyProductionByYear. Execute this query and here is the pivoted data. Once we have data in a table in the pivoted form, it's possible to use SQL to unpivot this data to get back to the original long form. Here is an unpivot operation where I select the state, year, and produce value from dbo.HoneyProductionByYear. Use the unpivot keyword to unpivot this data, get the produce value, and the individual years, 2008 all the way through to 2012. Execute this query and you'll see from the results that the data has been unpivoted and is now available in the original long form.

Summary

And with this demo, we come to the very end of this module where we use the structured query language to perform combining and shaping operations on our data. After being introduced to the Azure SQL database, that is Microsoft's managed service on the cloud, we saw how we could create and configure a database for us to use on the Azure portal. The data that we wanted to work with was present on our local machine. We uploaded it to a blob storage container on the cloud and saw how we could load data into our SQL database using Azure Data Factory pipelines. These pipelines can be used to integrate data from disparate sources, whether or not the sources belong to the Azure cloud platform. We then moved on to performing aggregation operations using the structure query language. We then saw how we could combine data stored in different SQL tables into one result by performing join operations in SQL. We studied the inner join, the left join, the right join, as well as the full join. And finally, we also used SQL to perform pivot and unpivot operations on our tables. In the next module, we'll move on from SQL to work in Python. We'll see how we can use Python libraries to combine and shape data, perform data cleaning operations, deal with missing data, outliers, and imbalanced data.

Combining and Shaping Data Using Python

Module Overview

Hi, and welcome to this module on Combining and Shaping Data Using Python. If you want to perform data modeling and analysis and you want to do so in a robust manner, yet in a way that allows for fast prototyping, well, Python is your friend. Python, as you know, is very simple and intuitive to use and offers a host of data science libraries that you can use to work with data. In this module, we'll start off by discussing some of the data cleaning and preparation techniques that you might use when you're working with data in the real world. Data in the real world might have fields missing or might contain outliers, and you need to be able to cope with both. In this module, you'll see how you can use the pandas library in Python to fill in missing values. We'll cope with missing values using listwise deletion, where we just drop records with missing fields, or we'll use imputation to fill in missing values. In this module, we'll also work with outlier data. We'll first see how we can identify outliers using box plot visualization and Z-scores. We'll also see techniques to cope with these outliers. And, finally, we'll talk about how unbiased samples make certain kinds of data modeling hard. We'll work with a hands-on demo where we'll improve the performance of a classifier machine learning model by rebalancing our dataset.

Data Cleaning: Missing Data and Outliers

When you're using data from the real world for analysis, you'll find that data is not often in the form that you wish it to be. Before you can actually get the data into the right form, there are a number of preprocessing steps that you'll need to apply, starting with data cleaning. Data cleaning encompasses a whole host of operations we'll talk about too specifically. You might have to deal with missing data in your datasets. When you collect data from a variety of sources in the real world, it may be possible that there are certain fields or information missing for every record. If you're performing data modeling using this data, how do you deal with missing data? That's what we'll discuss here briefly. Another issue that you might encounter when you're dealing with data in the real world is the presence of outliers in your data. Now these could be genuine outliers, which means they contain information, or they could be erroneously recorded points. Let's discuss the problem of missing data first and understand what techniques you have at your disposal to deal with missing data in the real world. Now there are two things that you could do if you have records with missing fields, deletion or imputation. Deletion involves dropping records with missing fields. Imputation involves filling in missing fields with values that are reasonable. The simplest way to deal with missing data is to simply ignore it and drop it from your analysis. Deletion is also referred to as listwise deletion. Here you'll delete an entire record or a row if a single value in a column is missing. This is simple, but it can lead to bias. Because of how simple it is to use in practice, you'll find that listwise deletion to be with missing data is the most commonly used method in practice. However, this can greatly reduce the sample size that you have to work with for your analysis. If you don't have much data to begin with, deletion can further reduce the amount of data that you use for data modeling. Also, if there is some kind of pattern in the records that contain missing data, if the values are not missing at random, then simply dropping records with missing fields can introduce a significant bias in your data modeling, leading to wrong results. In such situations, you might want to use imputation to delete missing values. Imputation involves filling in missing column values rather than deleting records with missing values. And there are a range of techniques that you can use to fill in missing values using imputation. These methods range from very simple to very complex. One of the simplest techniques of imputation is to replace missing values with a fixed value. The fixed value can be 0, infinity, or the column average. Using the column average has the advantage of not changing the mean of your dataset. Another technique to fill in missing values is to use interpolation. You can interpolate the missing values from nearby values. But if the values in the missing field are critical and they can be derived from other fields in your data, you can even build a model to predict missing values. Data cleaning and preparation in the real world also involves coping with outlier data. So what is an outlier? A data point that differs significantly from other data points in the same data set. An outlier is a data point that is unusual in some way. So what do you do if you have outliers in your data? You first need to be able to identify them and then cope with them. Identifying outliers can be done using two techniques, distance from the mean and distance from a fitted line. If you're using the distance from mean technique, you might convert your data to z-scores, where every value is expressed in terms of standard deviations from the mean. And you might mark as outliers all data points that lie, say, more than three standard deviations from the mean. Or you might fit a model to capture relationships in your data and any point that does not satisfy the existing relationship is considered an outlier. Once you've identified the outliers in your data, there are different methods that you can use to cope with them. You can simply drop the data points, which are outliers. This you'll do if you feel that outliers represent errors in your data. You can cap or floor outliers to specific maximum and minimum values, or you can set outliers to be equal to the mean value of the column.

Getting Started with Azure Notebooks

In this demo, we'll see how we can combine and shape data using a high-level programming language, that is, Python. The IDE that we'll use to write a Python code is Azure Notebooks. These are Jupyter notebooks posted on a VM instance on the Azure cloud platform. Go to notebooks.azure .com and sign in with your Azure subscription. The same account that you use to sign into your Azure portal to access other resources, cloud.user @ loonycorn.com. It's a personal account. Specify your password and get signed in. All of your Azure notebooks live within a project. If you don't have any projects, you can create a new one by clicking on this link. I'll give this project the name of this course, combining_and_shaping_data and Azure will assign a project ID. Click on Create and your project will be created. You can click through, and here is where we'll create folders and subfolders and create our Python notebooks. We'll have all our Python notebooks live within a code folder. Click through to code and create another subfolder here that will hold our datasets. This datasets folder will lie nested under our code folder. Click through to datasets and let's upload the CSV files that we'll work with. All of these CSV files are present on my local machine. Click on the upload option and choose From Computer. Click on the Choose files button here on this dialog to bring up an explorer or finder window where you can navigate to the files that you want to upload here of all of the CSV files that we'll use. Click on Open and click on Upload to load this onto the Azure cloud. Once the files have been uploaded, let's navigate back to our code folder. Within the code folder is where we'll create our Azure notebooks that we'll use to write code. Click on this drop-down and select Notebook. Azure notebooks give you the same interactive environment that you have with Jupyter notebooks on your local machine. I'm going to write all of the code here in Python 3.6, so that's the kernel I've chosen. Selecting this notebook will open it up in a new tab. And there you have it, your browser-based IDE interactive shell where you can write Python and view results right away.

Combining and Shaping Data Using Pandas

All of the data science libraries that you'll typically work with are already pre-installed on Azure notebook, so you can get started right away importing the libraries that you need. As with local Jupyter notebooks, you'll write your code and hit Shift+Enter to execute this code. The first dataset that we'll work with is the mall customers dataset, the original source of which is here on kaggle. Use pd.read_csv to read the contents of the CSV file into a pandas dataframe. Let's take a look at a sample of this data. You can see that it contains customer ID, gender, age, and the annual income of mall customers. The annual income seems to be specified in thousands of dollars. If you take a look at the shape property of this data frame, this will tell you how many records are available. There are a total of 150 records and there are 4 columns for each record. I'll now read in the contents of another CSV file that I have in here, mall_customers_score.csv. This is a different pandas data frame. This contains details for the same customers as in the previous data frame, but it contains spending score information. Spending score is just a rating from 1 through 100 for every customer of this mall. You can see that the customer ID column here is common for both of the data frames that we are working with. The shape of this data frame shows you that this too has 150 records, the same customers. Well, working with two data frames is tough. Let's get all of this data into one data frame using the pd.merge operation. Pd.merge is the equivalent of a join operation that you perform on a SQL table or an Excel spreadsheet. We are joining the data in mall customers info and mall customers score data frames on the customer ID. Invoke the head function on the resulting data frame, and here you see it, we have all of the data for customers joined together into a single frame. This was a row-wise merge operation. If you take a look at the shape of the resulting data frame, we have 150 records, the same number as before, but with 5 columns associated with each record. Let's say I get in some more data for new customers of this mall. This is present in the CSV file customers_data_2.csv. I'll read the contents of this file into a new data frame, and you can see that customer ID here starts from 151. The columns are exactly the same as our joined data frame from earlier, but these are new customers and there are 15 new records available. Well, it makes sense to have all of our customer data together in a single data frame. Let's do this using a pd.concat operation. Pd.concat allows you to concatenate or perform a union operation on the contents of two data frames, and here is the resulting data from, which includes a record from both of the original data frames, which we concatenated. You can see that the pandas index values in the very first column are all messed up. Call the reset_index function and replace the index inplace. And after having successfully combined all of our data together, our combined data frame has 200 records with 5 columns.

Identifying and Coping with Outliers

The dtypes property on a data frame will show you the data types associated with each column. Other than the gender, all other columns are integer values. The describe function on a data frame will give you a quick statistical overview of all of the numeric features. You can see the mean, median, standard deviation, and the percentile values for each feature. Observe that the mean for annual income is only about 59, 000, but the highest value is 170, 000. This seems to indicate that there may be outliers in annual income present in our dataset. I'll now use pandas to write out this combined customer data to a new CSV file, and we'll use it directly later. Let's continue exploring our dataset. Isnul.any will tell us whether there are any missing values for these fields. There are no missing values. Let's see the number of unique values for each field. As you can see, there are 200 unique customers and 2 unique values for gender. Pandas offers you built-in functions to visualize your data. It uses the matplotlib library under the hood to display charts. Here is a histogram of the annual income scores for all of the customers in our dataset. You can clearly see that there are a few customers who are outliers in annual income. Boxplot is a specific visualization type that's extremely useful in identifying outlier data. Invoke the boxplot function on your pandas data frame, and this will plot a box plot for every column of your data. In order to understand what a box plot represents, let's look at a box plot for just a single column, that is, the annual income. We know that there are some outliers here. Invoke the box plot function on your Pandas data frame and specify the column that you want to visualize. Here it's the Annual_Income, and here is the box plot. The box plot is a single visualization that gives you a bunch of statistical information about your data. This horizontal line that you see here at the center is the average annual income. The bottom of the box is the annual income at the 25th percentile and the top edge is the annual income at the 75th percentile. The size of the box here represents the interquartile range, and finally, these whiskers at the very top and the very bottom represent 1.5 times the interquartile range, as measured from the 75th and 25th percentile values. Now, any data point that lies outside of the range of these whiskers are considered to be outliers and are plotted separately. These customers seem to have an annual income of over $125, 000. Let's check who these customers are. Index into your data frame and access all of the records with annual income greater than 125, and here are the 2 records. These customers are clearly outliers in their annual income. Let's find a more robust way to identify these customers. Let's calculate the 25th percentile and 75th percentile values in our dataset and assign these to the variable Q1 and Q3. Print these out; 25% of the customers have annual income less than $40, 000 per year, and 75% of them are less than $77, 000 a year. The interquartile range is Q3 - Q1, that's equal to 37.25. Let's set minimum and maximum to be 1.5 times the interquartile range away from Q1 and Q3. You can see that minimum is a negative value, so we can ignore the minimum range, and maximum is equal to 133, 000. I'm going to add an additional column to our data frame that indicates whether a particular record is an outlier in annual income or not. I'll initialize all values in this column to false first. I'll then run a for loop iterating over every record in my data frame. For every record, I'll check whether annual income is beyond the maximum. This is what I'd consider an outlier. And if yes, I'll set Annual_Income_Outlier to be true. Let's run a sum operation on this Boolean column. This will tell us how many of the values are true. So there are 2 outliers in annual income in our records. One way to deal with outlier data is to simple replace our outlier values using the mean. Let's calculate the mean annual income. For the sake of simplicity, I've included the outliers annual income in this mean calculation. The mean income is around $60, 000. Once again, we'll run a for loop through our data, and if a particular row is an outlier where Annual_Income_Outlier is equal to true, we'll set the annual income for that record to be equal to the mean. Mean substitution is one possible way for you to deal with outliers. Let's take a look at the box plot of the Annual_Income once again, and this time you'll see that all outlier data have been substituted by the mean value of this column. There are no outliers in this box plot.

Detecting Outliers Using Z-scores

In this demo, we'll see how we can detect and identify outliers using z-scores, that is, all values expressed in terms of standard deviation from the mean. Here we are on a brand-new Azure notebook. Set up the import statements for the libraries that we'll use, pandas, numpy, matplotlib. I've turned off warnings here because there are certain operations which involve numeric conversations that through warnings that interfere with the demo. Rest assured, these warnings are safe to ignore. We'll continue working with the combined_customers_data that we had saved out earlier. Read this in into a data frame called customers_data_com. We are already familiar with this dataset, having used it in the previous demo. It's the same data. We also know that there are certain customers in this dataset whose annual income can be thought of as outliers. If you view a box plot visualization of the annual income, you'll see that there are two customers with very high incomes. These are outliers. Both of these customers have annual incomes greater than 125, 000. Let's identify these customer records. Here are the two customers whose annual income can be considered to be an outlier in our dataset. In order to identify these outliers in terms of standard deviations away from the mean, I'm now going to convert all annual income values to z-scores. This process of conversion to z-scores is referred to as standardizing your dataset. You can, of course, perform standardization manually by applying the mathematical operation, but even better, let's use the StandardScaler object from the scikit-learn library. Instantiate the standard scaler and invoke the fit_transform method on our annual income data. Fit_transform will standardize all numeric features that you specify as its input by subtracting the mean from all values in our dataset and dividing by the standard deviation. This will cause every value to be expressed in terms of z-scores or standard deviations away from the mean. The input that we pass into this fit_transform method is the annual income in terms of a numpy array that has been reshaped to be a 2D array. If you take a look at the shape of the array of this standardized result, you can see that it's 200, 1. That's what this reshape has done. Let's assign the standardized value of annual income to a new column in our data frame. I'll call this Scaled_Annual_Income. These are our annual income values expressed in terms of z-scores. Values expressed in terms of z-scores have a mean value very close to 0 and a standard deviation of 1, and that's what you see here, when you call describe on your Scaled_Annual_Income column. Let's now visualize the scaled annual income using a box plot visualization. As you can see, our records still contain outliers, and you can see that the mean of this box plot is at 0. Observe that all of the other values have been standardized so that they are expressed in terms of number of standard deviations away from the mean. If values slide below the mean, their z-scores are negative. If they lie above the mean, the z-scores are positive. Now that we have z-scores, what's an outlier? That's up to you to define, but typically any value that lies greater than three standard deviations from the mean can be considered an outlier. I've used the np.where function from the numpy library to find all records where the Scaled_Annual_Income is greater than 3. And you can see here that the indices of these records are 198 and 199. Let's take a look at the actual records in our pandas data frame, and here are the two customers that we know are outliers in annual income. Observe that the Scaled_Annual_Income is 3.8 and 3.9, greater than 3 standard deviations away from the mean. This time, let's assume that these outliers are erroneous points in our set and drop the records which contain this outlier information. We now have a data frame where all outliers in Annual_Income have been dropped. Let's see a box plot visualization of the annual income here, and you can see that the outliers have disappeared.

Handling Missing Values

In this demo, we'll see how we can use Python to handle missing values in our data. We'll do this demo on a brand-new notebook handling missing values. In order to see how we can work with missing values, I'm going to set up this little toy data frame. It contains two columns, column 1 and column 2, and you can see that there are some missing values represented by np.nan in column 1. Column 2 does not contain any missing values. An easy way to check whether your data frame has null or missing values is to use the isnull function. The result is a data frame of Booleans, which has the Boolean true for every missing value. If you'd rather just count the number of missing values in each column, you can call df.isnull .sum. This will give you a column by its result. Column 1 has 3 missing values, column 2 none. Pandas has this very useful function on the data frame called fillna that allows you to fill in missing values. There are different techniques that you can use to fill in missing values. The pad method simply carries the last observation forward. So for every missing value in a column, the previous value in that column is simply carried forward to fill in this missing value. Because we specified pad with limit 1, only 1 missing value was filled in, carrying the last observation forward. The second missing value was left as is. If you want two continuous values to be filled in with fillna, you specify limit is equal to 2, and both of our missing values have been filled in here in this result. Let's look at another technique to fill in missing values; the backfill, or bfill method. This will take the next observation and propagate that backward to fill in missing values. For record 1, which has a missing value, we filled it in with the value available in record 2. Just like the backfill, we have frontfill, or ffill, which carries the last observation forward. By default, frontfill has no limited and all missing values will be filled in by the last observed value. Instead of using imputation, let's say you wanted to use listwise deletion to get rid of missing values, you can simply invoke the dropna function. Axis=0 will drop records. Here in this result you can see that all records which had missing fields have been dropped. If you specify axis=1 for your data frame, this will get rid of columns with missing values. You can see that column 1 has disappeared and we are left only with column 2, which has no missing values. You can also drop columns with missing values if the number of missing values in a column exceeds a certain threshold. I'll specify the threshold using the thresh input argument. Shape of 0 will give us the number of records in a column. This will preserve a column only if 90% of the data is present in that column. Here you can see that column 1 has been dropped, column 2 has been preserved. This is because 90% of the records in our dataset is equal to 6.3 records. Column 1 has just 4 valid records; 3 are missing. It did not meet the threshold bar, so column 1 was dropped. The fillna function also allows you to fill in missing values with any value of your choice. Here we've performed mean substitution to fill in missing values. All missing values here are equal to the average value of this column. Pandas also makes it very easy for you to use interpolation to fill in missing values. In mathematics, interpolation is a method of construction new data points within the range of a discreet set of known points. Here are the results of interpolating to fill in missing values for this particular data frame. What if you want to replace your missing values with 0? Another way to do it, other than fillna, is to use df.replace and replace all nan values with 0's. I'll now construct a new toy dataset, which in addition to containing nans also contains infinity and negative infinity. Column 1 contains values that are negative infinity, as well as positive infinity. Column 2 contains just a negative infinity value. Let's see how many values are nans, or not a number, in our data frame. You can clearly see from this result here that negative, as well as positive infinity are valid numbers in Python. You can perform sum operations and pandas will treat negative and positive infinity correctly. The sum of the first column here is NaN because of the presence of missing values in our dataset, and the second column has a sum of -inf, which is correct. If you want pandas to treat infinities as errors, or not a numbers, you can set its mode to use inf_as_na = True. Now if you try to calculate the total number of missing or na values in your data frame, you'll find that pandas treats infinities as not a number.

Cleaning Data

In this demo, we perform a few different data cleaning operations on a dataset from the real world. Here we start off on a new Azure notebook called CleaningData. The dataset that we'll use here is the automobile miles per gallon dataset, available here on the UCI page. Read in the contents of the CSV file into a data frame called cars_data. This contains the model of the car, the miles per gallon, cylinder displacement, horsepower, weight, acceleration, and so on. There are total of 394 records in this data set and 12 columns. Let's first explore our data before we move onto data cleaning. I'm going to plot a scatter plot of displacement versus acceleration. Displacement is a measure of how powerful the car's engine is. I'm also going to color the data points of the scatter plot based on the horsepower of the car. The first thing that should strike you from the scatter plot visualization is the fact that the acceleration of the cars fall as their displacement rises. Also, the color of these data points give you an additional bit of information. Cars that have a high horsepower have lower acceleration and higher displacement. I'll now take a look at another scatter plot to view the relationship between two different variables. This time it's the weight of the car and the miles per gallon of the car. Once again, this is an inverse relationship; as the weight of the car increases, its miles per gallon falls. That makes sense intuitively. Let's use the isnull sum function on this dataframe to see whether there are any missing values in this dataset, and there are a few records where we don't have miles per gallon information, a total of 9 records. Rather than dropping these records, I'm simply going to use mean substitution to fill in these missing values. Calculate the mean for the miles per gallon column and use the fillna function to fill in missing values. Use the isnull sum function to check whether all missing values have been taken care of, and yes, indeed, they have. When you talk of data cleaning in the real world, it goes beyond missing values and outlier data. For example, you might want to get rid of some columns in your dataset because you want to just work with the information stored in the other columns. The drop function on your pandas data frame will allow you to drop columns that you're no longer interested in. If you look at the resulting data frame, it does not contain the bore, stroke, or compression ratio columns. They have been dropped. Let's take a look at one more way that you could clean data. The year column if you see is not of type int; instead, it's of type object. This is because some of the cars in our dataset have multiple years listed. Cleaning this data so that this year column represents an integer that is meaningful is a data cleaning operation that is very specific to our dataset. Depending on the kind of data you're working with, you might have some specific needs as well. Pandas is a pretty amazing data analysis library. You'll find functions that meet all needs. I'm now going to convert the year column to a string and check to see which of these strings have numeric values. You can see that 357 records have numeric values for the year; 37 do not. Let's see a sample of those records that have non-numeric year formats. Isnumeric = False. This code finds all of the records in our data frame that matches this condition. Exploring your data in this manner will allow you to detect patterns that will help you with data cleaning. You can see that the first four characters of each of these values are numbers. I'll now apply a data cleaning operation to extract just these first four characters, and I'll do this using a regular expression. Convert the year values to a string, call extract on the string, and extract the first four numbers. If you take a look at this extracted data, you'll see that they are all integers. We've extracted the first four numbers with the year field. I'll now convert this extracted column to numeric form by calling pd.to_numeric and assign this to the year column. The data type for the year column is now an int64. Now that we have the year in a numeric format, you can use this as a part of our data analysis and visualization. I'm going to plot a scatter plot of the year versus miles per gallon. And this scatter plot makes it pretty clear that newer cars tend to have better mileage.

Working with Imbalanced Data

Data that you work with from the real world may not always be sufficient for your analysis. And that's when you'll use oversampling and undersampling to get a more balanced dataset. When you're exploring data and inferring insights about the larger world from your data, inferring and deducing details about the population as a whole from a sample. The population refers to all the data out there in the universe. It's never possible to have all of the data with you for analysis, which is why you work with a sample of data. This is a subset of data that is available with you for exploration and analysis. Hopefully it's a representative subset so conclusions that you draw from the sample will apply to the population as a whole. There are several techniques that you can use to sample your population to get a representative sample, and these are hard to implement in practice, and it's possible that the sample that you end up working with is a biased sampled. And this is something you need to be extremely conscious of when you're working with data. For most kinds of analysis that you perform, you might want to work with a representative sample. But sometimes these unbiased samples make things hard. Let's say you're responsible for conducting a study to measure the health effect of a certain chemical. Now, exposure to the chemical is random and is extremely rare. You might find that in our dataset, you don't have very many individuals who have been exposed to this chemical. For a meaningful test, an unbiased sample would need to be huge to capture this rare effect of the chemical. And really, it may not be practically possible to get an unbiased sample that is this large. We might be constrained in the real world to working with smaller data sizes where we don't have that many instances of the kind that we are looking for. Our analysis might take the form of case studies where we only study those individuals who've been exposed to this chemical. Let's consider another example where working with unbiased samples make things difficult. Let's say you're building an image classifier model, which looks for photos with the Hawaiian crow. Now, the Hawaiian crow happens to be one of the rarest birds on earth, but it looks a lot like the common crow, which, as you know, is very common. So you're very excited about this classifier model that you're building and you train it with millions of images. You've called these images from all over the world. Now you're training corpus has millions of images with the common crow in it, but only a dozen images with the Hawaiian crow. The Hawaiian crow is rare. This is an unbiased sample, but it doesn't really work well for your classifier model. Your model just may not have enough data to learn about the Hawaiian crow with such few images available. Is it possible that we can reuse images of the Hawaiian crow? And this is where we'll use techniques like oversampling and undersampling. These are techniques that intentionally add bias to the data in order to make it balanced for your model. You tend to balance your datasets by oversampling uncommon x or y values. Values that are rare are oversampled so they have a more balanced representation in the result. And you'll undersample common x or y values. You'll reduce the number of common values so that it's more evenly balanced in the result. When you use forcibly balanced datasets to build your model, you have to be aware of a few pitfalls. Oversampling and undersampling will tend to reduce the accuracy of your model, but can increase the precision and recall. Precision and recall can be thought of measures of how well your classifier model is able to make positive identifications for unusual cases. Other techniques that are related to forcibly balanced datasets are the use of case studies and stratified sampling of your data. Stratified sampling is a technique where the total population is partitioned into subpopulations before it's sampled.

Handling Imbalanced Data with Scikit Learn

In this demo, we'll work with imbalanced data using the scikit-learn library. We'll build a simple classifier model, see how it performs on the imbalanced dataset, and then rebalance this dataset using oversampling and undersampling and see how our model improves. Here we are on a brand-new Azure notebook. The dataset that we'll work with is the diabetes dataset available here on Kaggle. I'll read in the contents of this dataset into a data frame called diabetes data. This dataset contains a bunch of information on Pima Indians and whether they've been diagnosed with diabetes or not. The diagnosis is the last column. This dataset has a total of 533 records and 9 columns. Let's explore the data that we have available. I'm going to plot a histogram of all features or variables that are present in this dataset. Here are all of my histograms, but the really interesting one here is the diagnosis. When diagnosis is equal to 1, that's when a person has been diagnosed with diabetes. It's pretty clear here that very few individuals in our dataset have been diagnosed with diabetes. This is a highly imbalanced dataset. We can confirm this by taking a diagnosis count of how many people have diabetes and how many don't. Diagnosis count counts the number of records which belong to class 0 and class 1. Class 0 means no diabetes, class 1 means the person has been diagnosed with diabetes. Print out the ratio of records and we'll also plot this using a bar graph. And this confirms the fact that this is a skewed dataset; 487 individuals don't have diabetes and only 46 do. And this histogram here is a visual representation of how this dataset is skewed. I'll now use this dataset directly to train a machine learning model, a classifier model to predict whether an individual has diabetes or not. The y labels for this model come from the diagnosis column, and the x features are all of the other columns. Let's split our dataset into training and test data. Training data is what we'll use to train our classifier and test data is what we'll use to evaluate our classifier. We'll build a very simple classifier here using the LogisticRegression estimator object. This is one of the simplest classifier models that one can work with. Instantiate the logistic model and call fit on your training data. Once we have a fully trained machine learning model, we can use it for prediction on the test data. Call logistic_model.predict and pass in x_test. Let's visualize how many predictions of our model were correct using a confusion matrix. This is a simple matrix, which gives you the actual labels versus the predicted labels from our model. You can see that there was exactly one individual in our test data with diabetes, but that individual was not identified correctly by our model. I'll now use the metrics namespace in scikit-learn to evaluate our model. I'll first evaluate our model based on its accuracy. Accuracy is a measure of how many of our models' predictions were correct. And this model has a huge accuracy of 91.79 %. But for a skewed and imbalanced dataset such as this one, accuracy is not really a good measure of our model, so let's try precision. The position score is a measure of how many of the positive identifications made by our model, that is, people who have diabetes, were actually correct. And here you can see why our model is not that good. Position is 0. The one individual who had diabetes was not correctly identified. Another metric that you could use to evaluate your model is the recall. Recall is a measure of how many positive identifications in the original dataset were correctly identified by our model. And this, once again, is 0. Well, that's because the one individual in our test dataset who had diabetes was not identified by our model. Well, this model could certainly do with some improvement. Let's concatenate our training data once again to form one data frame and reset the index on this data frame. I'm going to extract two separate sets of records from the original training data, individuals who have diabetes and those who don't. We'll oversample the records in our dataset so that we have greater representation of diabetics using the resample function from sklearn. Invoke the resample function and pass in as an input argument all of the records of the diabetic patients in our training data. We'll resample the original diabetic records with a replacement so that the number of samples that we are left with is equal to the number of samples that we have for non-diabetic patients. This resample function is what will help us feed in a balanced dataset to our classifier model. Concatenate the records of the non-diabetic patients with our oversampled diabetic records. We are now going to feed in this oversampled data as training data to build our classifier model. We can confirm that this dataset is evenly balanced by invoking the value counts on the diagnosis column. Print out the value counts for diabetics and non-diabetics and plot this as a bar graph. Our training data contains 487 records of non-diabetics, that is our original data, and you can see that it has 480 samples of diabetics as well. This is our oversampled data. The bar graph also shows us that our training data is now evenly balanced. Let's fit a logistic regression model this balanced dataset. Set up the training features and training labels for our model. Instantiate a logistic regression model and call fit on the training data. Once this model has been trained, let's call predict on the test data and let's plot this result in a confusion matrix. Well, our model seems to have performed better. In our test data, there were 23 individuals with diabetes; 9 of those were correctly identified by our model. Let's calculate metrics on this model. You can see that the accuracy of this model has fallen. Our model has clearly got more predictions wrong, but if you take a look at the precision score, it has risen to 28%. Of the individuals marked as having diabetes, 30% were rightly identified. That's a huge improvement over 0. Let's take a look at the recall score as well, and here, too, you can see a significant improvement. It's now 0.9. Of the individuals who actually had diabetes, 90% of them were correctly identified. I'll leave you with a small exercise that you can perform on your own. Resample the original dataset so that you undersample the non-diabetic individuals. Fit the logistic regression model once again and see how that performs.

Summary

And with this demo, we come to the very end of this module where we worked with Python and its assorted libraries to combine and shape data. We first discussed in some detail data cleaning and preparation techniques that you might use with real world data. You have to learn how to cope with missing data, as well as outlier points. We discussed two broad techniques to deal with missing data where you drop records with missing fields, or you fill in missing values using imputation. We then used the pandas library in Python to study the different techniques available to fill in missing values. It's possible that the points in the dataset that you're working with contain outliers. Now, outliers could be genuine outliers, or erroneous information. You first need to be able to identify these outliers before you can cope with them. We discussed different techniques for outlier identification, such as box plot visualizations and z-scores. And we used mean substitution and capping and floating mechanisms to deal with these outliers. And finally, we got a little taste of working with imbalanced data in the real world. We fit a classifier model on an imbalanced dataset and a rebalanced one and saw the difference in its performance. In the next module, we'll switch gears once again and work with the Azure SQL data warehouse on the Microsoft Azure platform. We'll see how we can integrate data from disparate sources into the Azure cloud.

Integrating Data from Disparate Sources into a Data Warehouse

Module Overview

Hi, and welcome to this module on Integrating Data from Disparate Sources into a Data Warehouse. The sheer volume and variety of data that you have to deal with today means that all of the data is never available on a single source. In order to understand what it means to work with big data, we'll first understand and study the differences between transactional and analytical processing and when we would choose to use either of these. We'll briefly talk about relational databases and data warehouses. Relational databases are typically used for transactional processing operations where updates are frequent, whereas with data warehouses, you'll perform analytical processing to extract insights from multiple sources of data. In the world that we live in today, all of our data is never in one location or accessed from a single source. This means you need a way to integrate data from different sources into a single location so that you can run business logic to extract insights from your data. In this module, we'll perform hands-on demos where we'll build ETL pipelines using Azure Data Factory and load data from disparate sources into a single data warehouse.

Transactional and Analytical Processing on Azure

As we move onto big data processing, we deal with evermore volume and variety of data, which is why data warehouses have become so significant nowadays. In order to understand the role of data warehouses, you first need to understand the differences between transactional and analytical processing. Let's take the example of two individuals. John works in order management support and is responsible for tracking and delivering e-commerce orders on time. Anna works for the same company, but is a revenue analyst and is responsible for tracking and monitoring revenues. Here is an example of a job or an action that John needs to perform on a daily basis. There might be three customers who want to ship orders to a different address. John has to go in and update the address on shipments and re-route these shipments to the new address. On the other hand, Anna's managers might want to know whether the new TV ads that the e-commerce site ran this year were successful. And they might also want to know how they compare with past seasons. Anna might have to dig a little bit into the past and track customer signup data when the campaigns were run and see how signups compare against past campaigns. As you can see, John's requirements are immediate and urgent, and these are examples of transaction processing. Anna, on the other hand, requires access to long-term data. Her needs may not be immediate. But they might be fairly complicated and require access to data from many sources. Anna performs analytical processing. I'll now compare and contrast the requirements of transactional and analytical processing systems. Transactional processing ensures the correctness of individual entries of data. Analytical processing accesses data in large patches and is less concerned about individual entries. Transactional processing requires access to recent data from the last few hours or days. The requirements tend to be more urgent in nature. Analytical processing requires access to older data going back months or even years for analysis. Transactional processing requires frequent updates to your data, so correctness of updates need to be ensured. Whereas with analytical processing, you'll mostly just read data and not update it. Transactional processing is associated with immediacy, so you'll require fast, real-time access to your data. Analytical processing typically involves long-running jobs that produce complex results. And usually it's the case that transactional processing involves working with data that comes from a single source. Analytical processing involves extracting insights from data, which might need data from multiple data sources. Back in the day when data volumes were small, it was possible to have a single database system to meet both of these objectives, of transactional processing, as well as analytical processing. You had a single machine which held structured, well-defined data. It was possible to access records individually or access the entire dataset. No replication was required. Updated data was available immediately. But today when our data sizes tend to be huge, it's very hard to meet all of the requirements of both transactional and analytical processing using the same database system. Big data typically does not fit into the disk or memory of a single system. You need a distributed cluster of machines. Data can be structured, unstructured, or semi-structured. When you're working with big data, updates tend to be less important than read access and data is replicated across multiple machines on your cluster, which means that propagation of updates take time. Analytical processing systems have to deal with all of the attributes associated with big data. The sheer volume of data in petabytes or terabytes, the variety of data, the number and types of sources can be huge. And, finally, the velocity of data. Data can come in very quickly in streaming format, which is why you need different kinds of systems for transactional and analytical processing. Transactional processing is typically taken care of using a traditional relational database management system, whereas for analytical processing, you would choose to use a data warehouse. When you're working with cloud platform providers, such as Microsoft, Google, or AWS, you'll find that services exist for all of these systems. The Azure SQL database is the managed relational database service on the Microsoft Azure cloud platform, and this is ideal for transaction processing applications. Since it's a managed service, the administrative overhead tends to be very low. Now we've already discussed that you need a data warehouse to store structured data used for analytical processing and reporting. Data warehouses typically store transform data fed in from disparate sources. These transformations can be used to clean and aggregate the data, and data is fed into the data warehouse via ETL pipelines where ETL stands for extract, transform, and load. ETL pipelines are programs or scripts which have business logic to automatedly extract data from disparate sources, transform data to satisfy a schema, and then load that data into a data warehouse. You can create and deploy ETL pipelines on the Azure cloud platform using the Azure Data Factory. This is a managed service meant for building complex hybrid ETL pipelines that integrate data silos and can include Hadoop and Machine Learning transformations as part of the pipeline. And finally, if you're working on the Azure cloud, the data warehouse that you'll choose to use is the Azure SQL data warehouse. This is the flagship data warehouse offering that competes directly with Google's BigQuery and Amazon's Redshift.

Creating an Azure SQL Data Warehouse and Uploading Files to Azure Blob Storage

In this demo, we'll see how we can use the Azure Data Factory to get data from the Azure blob storage container to an Azure SQL data warehouse. In this module, we'll work with air quality data from three different cities. Here is the air quality data for Gijon. The name of the Spanish city is a little hard to pronounce, so please forgive me if I don't get it right. You can see that this dataset contains the station name, lat/long information, and a bunch of information about pollutants, such as SO2, NO, NO2, CO, and so on. We'll work with another air quality dataset from a different Kaggle source; this is the air quality in Taiwan. The columns here are different. There are some columns which are common, though. There is information about common pollutants, SO2, CO, O3, and so on. And finally, here is a third air quality dataset that we'll use. This gives us the air quality in Madrid. This dataset is from a different source, and you can see that its structure and columns are different from the previous datasets that we looked at. We'll use the Azure Data Factory pipelines to bring together all of these disparate datasets into a single Azure SQL warehouse. Here we are on the Azure cloud platform portal. Use the search bar. On the left navigation pane, click on the option to create a new resource. Here is where I'm going to create a new data warehouse. Select databases and under that select the SQL Data Warehouse. Creating a SQL data warehouse is very similar to how you would create a SQL database. Specify the name of the database, loony-csd-sqlwarehouse, specify the resource group, and your subscription. I'm going to place the SQL data warehouse in the same server that we had created in an earlier module, the loony-csd-sqlserver located in US West 2. Click on the Create button to create this data warehouse, and within a few seconds, under a minute at least, you'll find that your data warehouse will have been successfully created. Select the Go to resource option, and here we are in our warehouse. Just like the SQL database, the SQL data warehouse also comes with a browser-based query editor. Select the editor and log in using SQL server authentication. Once again, we can use the same SQL authenticated user, the loony-csd-admin that we had created earlier when we worked with the Azure SQL database. Once you've successfully logged into the query editor, I'm going to minimize this pane on the left so that I have more room for my query editor. I'll now create a table called TaiwanAir in my SQL data warehouse. The TaiwanAir table has the site ID and a bunch of polluted information, CO, SO2, NO, NO2, O3, PM10, and PM25. Click on the Run button to execute this query, and this table will be successfully created. The object explorer on the left shows us that the TaiwanAir table has been successfully created. Let's now run another Create Table query. This time we'll create a table to hold information about the Gijon air quality, StationID, and a bunch of polluted information. Execute this query to create this table in our SQL warehouse. I'll run a third create table query to create the MadridAir table. Observe that this table does not contain a station ID or a site ID; it only contains pollutant information. Go ahead, execute this query, and create this table as well. All of these tables should now be available using the object explorer on the left. You can expand individual tables and view the columns that are available for each of these tables. You can expand each of these tables to confirm that the columns look like the ones that you set up. Everything looks good here. I'm going to use the left navigation pane and open up the storage accounts page on a new tab. Right-click and select Open Link in New Tab. Within the loonycsdstorage account that we had created earlier, we are going to upload data into blob storage. Select blobs and select the loony-csd-blob-container that we had created earlier. Here are the datasets that already exist here. Within this same container, I'm now going to upload the air quality datasets that are present on my local machine. Select the Upload option, bring up the finder dialog on your local machine, or your explorer window, whatever you choose to call it, and choose the Madrid and Taiwain .csv files and upload this to this particular blob container. Once these files have been successfully uploaded, we'll use an Azure Data Factory pipeline to load in the contents of this file into our Azure SQL warehouse. Hit Refresh and here are the two files that we just uploaded.

Loading Data from Blob Store to the Data Warehouse

Once our data has been safely uploaded to our blob storage container, let's head over to resource groups. Click on the loony-csd-rg and click through to the loony-csd-df that we had created earlier. I'm now going to author and monitor a new copy data pipeline, which will copy data using Azure Data Factory from my blob storage to my Azure SQL warehouse. Click on Next to configure the source for this pipeline. We already have the LoonyBlobStorageLink. That's what we're going to select here. And I'm going to use the Browse button to select the input file. Within the loony-csd-blob-container, let's load in data from the taiwan.csv file. We'll select a file, not a folder, within blob storage, so we can uncheck copy file recursively. Click on Next, and here is where we can specify our file format settings. Once again, we can accept the default settings as detected by this pipeline. Click on the Next button and let's move onto configuring our destination. We want to transfer data into an Azure SQL data warehouse, which means we need to create a new connection. Bring up the pane that allows us to create a new linked service. Select the Azure SQL data warehouse and click Continue. For consistency with my other links, I'll name this the LoonyAzureSQLDataWarehouseLink. A bit of a mouthful, I know. The subscription is a pay-as-you-go subscription. That's what my Azure account is. And the server is the loony-csd-sqlserver that we had created earlier. The database is the loony-csd-sqlwarehouse. Specify your SQL authenticated username and password, the loony-csd-admin, and the password, and then test this connection to see whether it works. And yes, indeed, it does. Hit Finish to finish configuring our destination. Select this SQL data warehouse link and let's move onto the next screen. We've already created three separate tables. We'll select the dbo.TaiwanAir table because this is the Taiwan air quality dataset. Hit Next and let's move onto configuring column mappings. The columns in our data warehouse table doesn't exactly match the input data, which means we'll need to reconfigure these columns. I'm going to use this drop-down so that county ID is in map to the CO column. I'll select the CO column instead, so that CO maps to CO. Once we made this change, you can see that Azure has auto-detected the other columns correctly. SO2 is mapped to SO2, NO to NO, and so on. Hit Next and let's configure the settings for this pipeline. There are a bunch of fault tolerance and advanced configuration settings that you can set up for your pipeline. You can choose the default or you can disable staging and PolyBase. If you use staging, your data might be stored in blob storage temporarily before it's moved into your data warehouse, and if you enable PolyBase, that is a more efficient copy mechanism allowed on Azure. I've chosen to disable both these. You can have it enabled if you want to. Hit Next and here is a summary of the copy that we are about to perform. Hit Next once again once you've reviewed all of the settings, and let's get this pipeline deployed. If you click on the Monitor button here, you'll be able to see the progress of this pipeline. Copy from blob to Azure storage is currently in progress. Wait for a little bit and it'll soon succeed. You can click on the Refresh button here to get the latest update. And now switch over to the tab that has the query editor and select the top 10 records from TaiwanAir. You can see from the results that the data has been successfully loaded into our data warehouse.

Loading Data from GCP Cloud Storage Buckets to the Azure SQL Data Warehouse

The reality today is you're probably using more than one cloud provider for your technology needs. So it's possible you have your data warehouse on Azure and that's what your business analysts use, but your data lies with several different cloud providers. In this demo, we'll see how you can get data that you stored within a bucket on the Google cloud platform into your Azure data warehouse. First start by uploading our data onto cloud storage buckets on the Google cloud platform, console.cloud .google .com. You'll need to have an account. You'll need to be signed in. When you're working on the GCP, all of your resources are provisioned within a project. Think of the project as equivalent to an Azure resource group. Here I am within a project called loony-csd-project. Use the hamburger icon on the top left to bring up the navigation menu that shows you all products and services available on the GCP. Head over to Storage and go to Browser, and here is where you can create and work with Cloud Storage Buckets. Click on the Create button option, and that'll take you to a page where you specify a globally unique name for your bucket. I've prefixed it with loony so that it's easy for it to be globally unique. You can then select the kind of bucket that you want to create. I'll go with a multi-regional bucket where my data will be replicated across multiple GCP regions, or geographical locations. Once the bucket has been successfully created, click on the Upload files button here. This will bring up an explorer window on your local machine, and choose the Gijon.csv file. So here are some of our air quality data scattered across multiple cloud platforms. Gijon.csv is on the GCP. We are going to use an Azure Data Factory pipeline to pull the contents of this file onto our Azure SQL data warehouse, and in order to enable this cross-platform access, the Azure Data Factory needs an access key and secret to access this cloud storage bucket on the GCP. Head over to the Settings page for this particular bucket, and here you'll find a tab for Interoperability. This is where we can configure interoperability across multiple cloud platforms. I'm going to make the loony-csd-project my default project so that my access keys are created within the default project. Click on the button here which says Create a new key. Now this key is secret. I've deleted this bucket so it's fine for you to see it. You'll get an access key and a corresponding secret. Make sure that you copy it somewhere. We are going to use it soon. We are now ready to configure an Azure Data Factory pipeline to read the contents of the CSV file from our GCP bucket. Click on the Copy Data option here in order to set up a new pipeline. Here we are within Properties. Let's call this pipeline CopyDataFromGCPBucketToAzureSQLWarehouse. The next step is to specify a connection to the source from where we want to copy this data. We're going to have to create a brand-new connection here because we are going to connect to a cloud storage bucket on the Google cloud platform. If you scroll down here on the options for a linked service that you can connect to, you'll find along with connections to AWS services, you can also connect to GCP services. Observe that Azure uses an S3 API to connect to Google cloud storage. Amazon has provided an API interface for cloud storage buckets, and that's what Azure uses. This might seem strange at first, but not really. All cloud platform providers understand that this is a multi-cloud world, and they've enabled these connections. Hit Continue to configure this linked service. I'm going to call this the LoonyGoogleCloudStorageLink. Scroll down and specify the access key ID. This is the access key copied over from the interoperability page on our Google cloud storage bucket. Also specify the secret access key as well and go ahead and hit Test Connection. And yes, we've successfully managed to connect to our GCP storage bucket. Click on the Finish button to finish configuring the source connection for our linked service. With this link selected, hit Next, and let's go on and select the file that we want to load in from our GCP bucket. Select the loony-csd-bucket on the GCP. Click through, and here is Gijon.csv. Hit Choose, and this is the file that we are going to load in using Azure Data Factory. It's a file and not a folder, so I'm going to uncheck the copy file recursively option. Click on Next. Here is where we specify the file format settings. We can accept the default settings that Azure has detected here for the file format, and here is the preview of the data that we are about to load in. If you want to view the schema that Azure has detected for this file, you can switch over to the Schema tab and see how all of the columns map. Click on Next, and let's continue configuring our destination data store. Well, we want this data to be moved into the SQL data warehouse that we had set up earlier. That's the destination I'll select, and hit Next. Let's map this data to the right table where we want the contents to be loaded. The GijonAir table is where the contents of this file goes. Select that table and hit Next. And let's configure the column mappings for the source data and our destination table. The columns are not really lined up correctly. Your station name is mapped to CO. Let's switch this around. Click on this drop-down here and select the CO column from the source database so that it goes to the matching table. For this particular dataset, it so happens that once I map the CO column correctly, Azure auto-detects the other mappings as well. So once you make sure all of them are okay, click on Next, and we'll move onto configuring the settings for this transfer. I'm going to disable staging. I don't really want it. You can choose to turn it on if you want to. And I won't use PolyBase. For the purposes of this demo, this is fine, but if you have a huge amount of data to transfer, enabling PolyBase will make your copying far more efficient. So click on Next here and let's move on and take a look at a summary of our transfer. The source is a GCP bucket accessed using the S3 API and the destination is the Azure SQL data warehouse. Everything else looks good. Click on Next and let's get this deployment up and running. Once the deployment is complete, you can monitor the progress of this copy data pipeline by clicking on this Monitor button here. This pipeline is currently in progress; it's not done yet. Hit Refresh and in a couple of minutes you'll find that this pipeline will have succeeded. I'm now going to switch back to the tab where my data warehouse query editor is open, and I'm going to run a query to see if the contents of my CSV file have been loaded in. Switch from TaiwanAir to GijonAir, this is the table that we populated in our data warehouse, and select the Run option to execute this query. And this table is no longer empty. Contents of our CSV file that we had stored on Google cloud storage buckets have been loaded into our Azure SQL data warehouse using the Azure Data Factory pipeline.

Enabling Resource Permissions to an Azure Active Directory App

In this demo, we'll see how we can use Azure Data Factory pipelines to perform data transformations. Specifically, we'll create a Spark activity that will allow us to create an on-demand Spark cluster to transform our data. And this data transformation using Spark will run as a part of our data factory pipeline. In order to run a Spark activity on an on-demand HDInsight cluster where HDInsight inside is Azure's managed Hadoop service on the cloud. You need an Azure Active Directory application and a service principal that has the right permissions to access your cluster in order to run data transformation operations. Head over to Azure Active Directory using the left navigation pane, and here is the page that will allow us to create an Azure Active Directory application and register it. On the center navigation pane, head over to App registrations, and this will take you to a page where you can register your app. The Spark code that we'll execute for data transformation will run under the identity for this app. Click on register an application and give this app a meaningful name. I'll just call it LoonyCsdApp. I want this app to be used only within my organization. Click on Register and wait for the app registration to be complete. The code that you write will run under this app, and when you have code that needs to access or modify resources, you need to create an identity for the app. And this identity is known as a service principal. Click on Home here so that you can create a service principal within your subscription. Here are all of the resources that we've been using recently. Observe that our pay-as-you-go subscription is available here at the bottom. Within this, go to access control, that is, identity and access management, and click on the Add button here to add a new rule assignment so that your service principal that is associated with the LoonyCsdApp has the right permissions to access resources on the Azure cloud. Add role assignment and select the role that you want to assign to your app. I'm going to go with the project Contributor role. This is a role-based access control, so every role comes with an associated set of permissions. We want to assign these permissions to a service principal associated with our app. Let's select the Azure Active Directory app to which we want to assign this role. This is the app that we just created, LoonyCsdApp. Once you've made this selection, go ahead and hit Save, and the service principal associated with the LoonyCsdApp will be added as a contributor to our subscription. Let's go back to our app registrations page. Go to Azure Active Directory and select App registrations. Click through to our LoonyCsdApp here. There is a bunch of information available on this app, which we'll need to use to configure our Azure Data Factory pipeline when we are transforming data using Spark. You can see here on this apps page that it has an application or client ID. We'll be using this. We also have a directory or tenant ID; we'll be using this as well. Our app now has the right permissions, but we'll need to set up a client secret so that our app can prove its identity when it's requesting our resources. We'll create a new client secret so that our LoonyCsdApp can prove its identity. Click on New client secret. Click on Add. I'll have this client secret be valid for a year. And here is our new key. Copy over the value associated with this key. We'll be using it later when we configure our Azure Data Factory pipeline. We've completed the preparatory steps for access to our HDInsight cluster.

Python Script to Transform Data Using Pyspark

Now let's take a look at the Spark code that we'll use to transform our data. This code uses pyspark in Python and is written in the transform.py file. This code is fairly straightforward. Obviously in the real world, your data transformation operations will be more complex. Observe that we instantiate a SparkSession to access the Spark cluster using the LoonyCsdApp, which has been authorized to access resources on the cluster. Our data transformation operation simply reads from the Azure blob storage container. We read the contents of the file that gives us the air quality for Madrid. From this file, we'll extract only those columns that we're interested in, which contains information about pollutants, such as CO, NO, NO2, O3, and so on. If there are any records with missing values for these columns, we go ahead and drop those records and write out the contents of our clean data back to Azure blob storage. I'll just scroll over to the right so that you can see that we are accessing the contents of the Madrid.csv file. The cleaned output data after it has been transformed will be stored in a folder called madriddata. The remaining code is just Spark template code. We call spark.stop to end the session, and we invoke this main function from the main Python module. Before we create our Spark activity, our Spark transform.py code needs to be uploaded to our blob storage container. Go to storage accounts and click through to loonycsdstorage. We'll use the same container which hosts all of our other CSV files. Click through to blobs, and here is our loony-csd-blob-container. Click through and select the Upload file option. The transform.py file is present on my local machine, so I'm going to select this Browse button here. This will bring up my finder or explorer dialog. Select transform.py and hit Open. Hit Upload so that your Python script is uploaded to your blob storage container. You can click through and take a look at the contents of your file. Azure also allows you to edit this blob file if you want to. Click on Edit blob and here is where you can edit your code right here within your browser. You can ensure that your Python script is setup correctly here. Observe here at the bottom that Azure has correctly detected that this is Python code, thanks to your .py extension.

Transform Data Using a Spark Activity

We are now ready to set up our Azure Data Factory pipeline, which will contain our Spark activity used to transform our data. Click on this little edit icon here on the left. This will show us all of the linked services available as a part of our Azure Data Factory. Click on New here so that we can start configuring a new link service that will allow us to perform computations on our data. So far we've only been working with data store linked services. Click on the Compute tab here, and this will show you all of the services that you can access from within Azure Data Factory to transform your data. You can use serverless lambda functions. You can use Azure Databricks, Data Lake Analytics, Azure ML, and Azure HDInsight, that's our choice. HDInsight is Azure's managed Hadoop service on the cloud. And this is the cluster that we'll use to run our Spark activity. Name this linked service, and you can choose the kind of HDInsight cluster that you want to use to run your Spark activity. I'm going to go with an on-demand HDInsight cluster. If you have a preexisting HDInsight cluster, you can use that as well. On-demand means that Azure will provision a brand-new cluster run by Spark code and then bring down the cluster once it's done. You now need to tell Azure where your Hadoop code is. Choose this drop-down and we already have a linked service set up to our LoonyBlobStorage. This is a link to our blob storage container where we've uploaded transform.py. So this is the selection we make. The next step is to specify the kind of HDInsight cluster that you want to create. I don't want a Hadoop cluster; I want a Spark cluster. We now need to identify the Azure Active Directory app that has permissions to access these cluster resources. Here is where you specify the service principal ID of our LoonyCsdApp. This is the application ID from the LoonyCsdApp page that we had looked at earlier. This service principal will prove its identity and authenticate itself to the cluster using this client secret. And when you specify the service principal ID of the LoonyCsdApp, Azure auto-populated the tenant ID for you. There is an option here to specify the prefix of the HDInsight cluster that you want to create. I'll just call it loony-csd-. Here is our pay-as-you-go subscription, which we are going to use to create this cluster. The resource group will be the loony-csd-rg exactly like before. And let's select the region where we want this cluster to be created. All of the resources that we've used in this course have been in the US West 2 region. Let's continue with that. Expand the OS type advanced settings. Here is where you can specify a username and password for your cluster. Here is our cluster username. Give it a password that you'll remember and let's move on. Hit the Finish button here and this sets up our linked service to an HDInsight compute cluster which is running Spark. Click on the plus button here on this page in order to create a new Azure Data Factory pipeline that will include our Spark activity to transform data. Observe that this is a more general purpose pipeline, not the standard copy data pipeline that we've been seeing so far. I'll call this pipeline TransformBlobDataUsingSpark. Azure allows you to set up a visual representation of your pipeline here on the center pane. So I'm going to expand this so that we have more room to work with. Select the HDInsight option here on the left. Here is where we can configure our spark activity. Select this activity using your mouse and drag it onto your center pane. This pipeline now has a Spark activity, and with this activity selected, you can bring up the configuration pane to configure the settings for the Spark activity. Give your activity a meaningful name. I've just called it LoonySparkActivity. Let's configure the HDI cluster for this activity. Click on the HD Cluster tag and let's select the linked service to the HDInsight cluster that we had set up earlier. The next step to configure this activity is to specify the script or the jar file that will run your Spark job. Use this drop-down here in order to specify the job linked service. Remember that the transform.py file that runs our Spark code is present in the LoonyBlobStorage linked service. We now need to point to the right script file within this blob storage. Click on Browse storage. Select and go through to the loony-csd-blob-container and select the transform.py file and hit Finish. Here is the Spark script that we want to run on the HDInsight cluster. You can choose to preview the script here if you want to just to reassure that we are indeed running the right transformation code. Everything looks fine; hit Finish and let's move on. The advanced option here will give us the configuration settings that we want to use to debug our script. The debug information that I want is on Failure. If the script fails, I want to be notified. This completes the configuration of our Spark activity. Let's go back and see if it has been set up right. Select the Spark activity and click on the Validate button, which will allow you to validate your pipeline. You can see that our pipeline validation is successful. No errors were found. Well, that's good news because now we can move onto publishing this activity. Click on the Publish All button here to the top left to publish your pipeline. Having deployed and published our pipeline successfully, we are now ready to kick start this pipeline. Click on the Add trigger button in order to get it started. After all of this configuration, I'm kind of impatient to see whether this works. So I'm going to select Trigger Now. This brings up a little message which tells you that the trigger pipeline will now use the last publish configuration. Hit Finish, and your pipeline has been started. Let's head over to the pipelines page and view all of the pipelines that are currently in progress. Well, the pipeline isn't really there here. This happens sometimes. You can hit Refresh, or, in my case, I have a filter set up. Click on the Filter option and get rid of all the filters that might prevent you from seeing the pipelines that you have running. Here is our TransformBlobDataUsingSpark pipeline. It's currently in progress. Wait for a little bit. Hit Refresh, and you'll find that soon this pipeline would have succeeded. Well, I spoke too soon. It's been running for about 14 minutes so far. It'll take awhile because it has to instantiate a cluster on demand to run your Spark activity. Hit Refresh once again and wait. The pipeline has disappeared. Well, that's because it has succeeded and is no longer in the In Progress tab. Click on the Succeeded tab and you'll find your TransformBlobDataUsingSpark pipeline. Well, you see the happy succeeded message here, but did the data transformation actually happen? Let's find out. Switch over to our blob storage account, loony-csd-blob-container. The pipeline output should have been dumped here. Hit Refresh and there you see it, our madriddata folder. Within this folder should be our transform data. Here are two files which contain all of our transform data. Let's click through and take a look at the contents of one of these files here. This is the file metadata. Edit blob will give us actual file contents. Click on Edit blob and here is our transform data output by our Spark activity. Our pipeline to transform data ran through successfully.

Copying Data from a Blob Storage Folder

Now that we have the transform data for MadridAir in our blob storage container, let's run a copy data pipeline on Azure Data Factory. This will copy data from blob to our Azure SQL data warehouse. These steps should be familiar to you, so I'll run through them quickly. Choose the LoonyBlobStorageLink as the source of our data. Browse to find the right folder which contains the files whose contents we want to copy into our Azure SQL data warehouse. Here we have it, the folder madriddata is what we are looking for. This time we want to copy the contents of all files that lie within this folder, so I'm going to leave the option checked for copy files recursively. Double-check the file format settings. Everything seems to be correct. Azure has auto-detected the file correctly. Let's take a look at a preview of our data. You'll find that our column headers have been lost. Column headers are now 1, 2, 3, and so on. Our column headers are, however, present in the first row of our data. That's fine. Our copy pipeline can take care of this. Hit Next and let's move onto configuring our destination. We'll copy to the LoonyAzureSQLDataWarehouse link and the table is MadridAir. Click on the Next button and we'll check to see whether the columns from our source data have been mapped correctly to our destination table. Well, they haven't, and we only have the column numbers to work with. This makes things difficult for us. You'll need to know what column number maps to what kind of pollutant. I note this, so I'm going to make a few changes, and then we'll compare with the original file so that we know that we have set things up correctly. If the same field in the original data is mapped to different columns of our SQL data warehouse table, you'll get a little error here as you can see onscreen. I'm going to go ahead and fix the remaining mappings. At the end of this remapping exercise, we'll compare with the original dataset to make sure that all of our mappings have been set up correctly. So you'll just have to accept my word for now that these mappings are indeed right. All right, my remapping of fields is now complete, and here is where I verify that all of the original fields have been mapped to the right column in the destination. Field 1 is the CO pollutant, field 2 is NO, 3 is NO2, all of these mappings seem to be correct. Hitting Next will get you to the page where you can configure the copy options for your pipeline. Everything looks good. Once again, I've disabled staging and disabled PolyBase. Hit Next so you can get to the page where you can review this pipeline before you deploy it. Hit Next, and this pipeline will be deployed. Once the pipeline has been deployed, the Monitor button here will allow you to monitor the progress of this pipeline. This pipeline has succeeded. It was a short one. Let's switch over to our query editor and run a query to check whether the MadridAir table has been populated. Hit Run and here is the data. We've successfully loaded our Spark transform data to our Azure SQL data warehouse.

Summary

And with this demo, we come to the very end of this module where we discuss how we could integrate data from different sources into a single data warehouse on the Azure cloud. We started this module off by first discussing the differences between transactional and analytical processing. Transactional processing typically involved performing urgent, immediate updates where updates have to be reflected right away in your database. We saw how analytical processing is critical to extract insights from our data and typically involve the use of data going back several months or years. We then discussed relational databases and data warehouses. Relational databases are optimized to meet the needs of transactional processing, whereas for analytical processing, you'll use data warehouses. Analytical processing involves bringing together data from disparate sources. And we saw hands-on demos of how we could do this using Azure Data Factory and ETL pipelines. We integrated data from Azure blob storage, as well as data stored on another cloud platform, the GCP. We also saw how we could transform data using Spark activities within our ETL pipelines. In the next module, we'll see how we can use the stream analytics job on Azure to ingest streaming data into our data warehouse, and we'll visualize this streaming data using Power BI.

Working with Streaming Data Using a Data Warehouse

Module Overview

Hi, and welcome to this module on Working with Streaming Data Using a Data Warehouse. Traditionally we've seen that all data was available up front so batch processing operations were sufficient. However, when you receive data in real time, you have to work with stream processing. We'll first compare and contrast batch and streaming data and the processing techniques that we use. We'll then talk about how we can perform aggregations on streaming data by defining windows over our stream. There are different kinds of windows possible, the tumbling window, the sliding window, the global window, and others. The notion of time is significant when you're working with streaming entities, and we'll discuss three types of time associated with streaming data, event time, ingestion time, and processing time. We'll then get hands-on using the Azure SQL data warehouse that we set up on the Azure cloud platform. We'll see how we can ingest streaming data with Stream Analytics, and we'll then visualize these real-time streams using Power BI. You'll get an idea of how easy it is to leverage Azure services to work with streaming data.

Batch and Streaming Data

A lot of the interesting information in today's world comes from data that is available in real time, that is, streaming data, and streaming data requires stream processing. Let's understand the differences between batch processing and stream processing. Batch processing refers to operations that you'll perform on bounded datasets. All of the data is known up front and they are processed in batches. Stream processing operations are the ones that you'll perform on unbounded datasets. They are processed as streams. All of the data is not known up front. When you're working with data, you don't have the entire dataset. Let's compare and contrast these two side-by-side. Batch processing refers to bounded finite datasets, and stream processing works with unbounded, infinite datasets. When you're working with bounded datasets using batch processing, there is a slow pipeline from data ingestion to analysis. You first collect all of the data, it's stored on disk somewhere, and then processed. With stream processing, processing is immediate. As soon as data is received, it's acted upon. With batch processing, the updates that you receive tend to be periodic in nature. There are several jobs that will process your data in parallel and you'll receive updates when those jobs complete. With stream processing, you get continuous updates as the job runs constantly. With batch processing, the order in which the data was originally important tends to be insignificant or unimportant. Even if there is an inherent order in batch data, all of the data is available up front. It's easy for you to order it before processing. With stream processing, however, the order in which the events are received is important, and out of order arrival is tracked. When you're working with batch processing, there is a single global state of the world that is known at any point in time. All of the data is available up front. You know exactly what you have to work with. With stream processing, there is no global state. You only have a history of all of the events that you've received so far. You have no idea what's going to come in the future and how that might change your analysis. Back in the day, we only worked with traditional batch processing systems. All of your data was stored in either files or on databases somewhere. Whatever data was available to your organization on reliable storage was considered to be the source of truth. However, nowadays batch data is no longer sufficient for our analysis needs. You might have historical records stored on reliable storage somewhere. In addition, you might need to process real-time events from streaming data, which is why the processing architecture that you need to work with has to be a stream-first architecture in order to meet the needs of analysis. Typically with this stream-first architecture, all of your data from reliable storage, as well as real-time streams, are passed into a buffer or a message transport system. The buffer is responsible for buffering and ordering the data. The output of this buffer is then passed onto the stream processing system. The most significant fact about the stream-first architecture is that the stream itself is considered to be the source of truth. When you're working with data streams, your stream may not be from a single source, which is why you need a buffer which takes an event data from multiple sources. The message transport has to be performant and persistent and not lose the data that comes in. You need the message transport to decouple multiple sources from your actual stream processing. Technologies that serve as message transport systems, as well as stream processing systems, are Kafka and MapR streams. When you use a stream first architecture, there are also requirements of your stream processing system. It should have very high throughput and low latency because the velocity of your streams might be very high. It should be fault tolerant with a low overhead. Events should be processed quickly, and it should be able to manage out of order events and reorder them so that they can be processed correctly. The stream processing system should be easy to use and maintainable and should have the ability to replay streams. Here is a big picture overview of the stream processing model. Data that you work with is received from a data source. There can, of course, be multiple sources of data. All of the data or events received by your stream processing system undergo transformations. Transformations refer to data cleaning, conversions, and aggregations that you want to perform on your data. And this transformed data will finally be stored in a data sink, which is reliable storage of some sort. The transformations that you apply to your streaming data is not just one monolithic operation. It is a series of operations that are chained together in the form of a graph. Transformations are typically directed-acyclic graphs where the nodes represent operations that are performed on your streaming data.

Types of Windows

Streams are unbounded; you don't have all of the data up front. When you perform aggregation operations on streams, you'll use windows. Let's talk about those. Let's first talk about the various kinds of transformations that you can apply to streaming data. Stateless transformations are those which are applied to only a single stream entity. On the other hand, you might want your transformations to have some kind of state. Stateful transformations are those which accumulate across multiple stream entities. Stateful transformations on your streaming data are also referred to as window transformations. That's because information is accumulated across a window in a stream. A window includes multiple streaming entities, and there are different ways in which windows can be defined. Let's see an example of stateless transformations on streaming data. Let's say you're on a busy highway and there is a radar that tracks the current speed of the cars that pass by. This radar will record the speed of each of the cars that pass by, and the speed is probably transmitted to some kind of command center where someone's monitoring these speeds. At any point in time, you're only processing the speed corresponding to a particular car. Each entity is operated on standalone. This is an example of a stateless transformation. Speed exceeded? Alert triggered. But let's say you're back on the same highway and you want a system which tracks how busy a particular highway is. As cars pass by, you keep track of the number of cars that have passed by at any point in time. This is a counting operation and it requires the knowledge of some kind of state. You might want to track the number of cars that pass by in a minute, in an hour; these are your time windows. We already know that streaming data is data available over a period of time. And a window is a subset of a stream based on either a time interval, a count of entities, or interval between entities. So a window could include all of the data that was received, say, in the last 10 minutes, or it could include 10 streaming entities at a time, that is, the count of entities, or the window could include all of the data received such that the gap between two streaming events is no more than a minute. That is the interval between entities. Once you've defined the window of data that you want to work with, you can then apply transformations to all entities within a window, and these are typically aggregation operations such as sum, min, max, average, and so on. There are five types of windowing operations that you can perform on streaming data. You have the tumbling window, the sliding window, the count window, the session window, and the global window. We'll discuss in some detail the tumbling, sliding, and global window, and briefly cover the count window and session window later on. We'll have our window operate on this stream of data that you see here onscreen. This is a stream of data, and the tumbling window applies a fixed window size to this stream. Observe that this size of window depends on time. So, the tumbling window could be 10 minutes or 10 seconds, but the time is fixed, making the window size fixed. Tumbling windows are so called because they do not overlap in time. Every window is discreet, and the number of entities in a window might be different. So you might have a window with three entities, another with four entities, it depends on how the streaming events come in. When you define a tumbling window over your streaming data, you can imagine that the window tumbles over your input stream in a non-overlapping manner, as you can see, visualized here. Once you've defined some kind of window over your streaming data, you can apply aggregation operations on entities that lie within a window. For example, you can apply the sum operation on each window. As the window tumbles over your streaming data, you'll perform the sum aggregation and get the sum of streaming entities in each window. Let's now discuss sliding windows over your streaming data. It's very similar to the tumbling window in that we have a fixed window size. However, the way the window moves over your data is different. There is overlapping time, and that is the sliding interval. The window size is fixed, but because of the existence of the sliding interval, a single entity may belong to more than one window, but the number of entities within a window will be different. It depends on how many events came in in that window of time. As the fixed size window slides over your streaming data, you'll find that there are certain data points that overlap from one window to another, as you can see visualized here onscreen. Once you've defined a window over your streaming data, no matter what that window type is, you can apply aggregations on all entities within a window. For example, let's apply the sum operation on each window, and you'll get results which look like this for this particular stream. We've understood the tumbling and sliding window. The global window is far simpler to understand. If you include all of the data that you've seen so far in this stream in one window, that is the global window. Let's quickly discuss the two other window types. The count window basically is a window over a fixed number of entities. You can see a count of three, a count of six. As soon as you receive six entities, that makes up a window. And the session window as its name implies is used to identify session. Let's say you have a user for your e-commerce site. As long as the user is clicking around, it's part of the same session. But if the user falls silent and does not click around for, say, a minute, you can say that the session has ended. All of the data that is received in a single session is part of one window. So this depends on the time interval between entities.

Watermarks and the Notion of Time

We saw in the last clip that in order to perform aggregations on your streaming data, you'll divide your streaming data into windows. Tumbling and sliding windows consider all entities which arrive at a fixed interval of time. And we've also repeatedly spoken of the fact that when you're working with streaming data, time is extremely important. When you're working streaming data, you'll find that there are different notions of time that can apply to entities in a stream. The three notions of time that are associated with streaming data are event time, ingestion time, and processing time. Let's consider each of these in turn, starting with event time. Event time refers to the time at which that streaming event occurred at its original source. This is time recorded even before the event has entered your stream processing system. If your event originated on your mobile phone or a sensor or your website, event time is the time recorded at this source. And event time is usually embedded within the records of your streaming data, and this is how stream processing systems are aware of event time. Event time is extremely important because this is what is used to give correct results of the right order of events in case events appear at your stream processing system out of order. The second notion of time that you'll deal with in stream processing is the ingestion time. This is the time at which the streaming event enters your stream processing system via a source. The timestamp for ingestion time is chronologically after the event time. The event occurs first and is then ingested. And ingestion time cannot be used to handle out of order events. And, finally, the third notion of time that you'll have to deal with is processing time. This is the system time of the machine which is responsible for processing your streaming entities. And this time is chronologically after the event time and ingestion time. System time tends to be non-deterministic in nature because you have no idea how long a particular event will have to wait in the buffer before it is actually processed, and you have no idea how long your processing operations will take. But working with system time is far simpler than working with either event time or ingestion time. It's simple and there is no coordination required between streams and processors when you use processing time. As far as streaming entities are concerned, time is significant. But you know that in the real world, nothing ever happens on time. So how late is late? Let's say you have class at 9:00 AM and class starts when the clock strikes 9:00. If someone comes in at 9:01, is that late? Realistically, there are definitely going to be a few students who are going to be a minute late. But what do you do if a student arrives over an hour late, at 10:10? Do you allow the student in or send them back? Now, you know that in the real world, the professor of your class knows what lateness is reasonable, and students who enter the class within this reasonable lateness are late, but the professor still allows them in. Students who enter after this reasonable lateness limit are too late and the professor might choose to send them back. So, there is a concept of allowed lateness. Now it turns out that stream processing systems work in exactly the same way. You can enforce a concept of allowed lateness using watermarks. The watermark is the system's way of knowing what lateness is reasonable, and this reasonable lateness can be configured by the developer of the stream processing system. Data which enters within this reasonable lateness is late, but okay, and can still be included in your aggregations. But data that appears at your stream processing system after this reasonable lateness is considered to be too late and not included in our computations. As a developer working on a stream processing system, you'll configure watermarks as the threshold of allowed lateness. Watermarks are specified in terms of event time. When you have late data come in, data which arrived within the watermark is aggregated into your processing, but if you have data that arrives after the watermark, it is dropped. It's not included in your processing.

Creating a Stream Analytics Job and a Power BI Pro Account

In this demo, we'll see how we can ingest streaming data using a Stream Analytics job on the Azure cloud platform, and we'll visualize this stream of data using a Power BI dashboard. Imagine that the data that I'm about to stream in comes from multiple sensors that I have set up across the world. I don't actually have these sensors. That's why we'll work with this data in a .json file. We'll upload this InputStream.json file to a blob storage container and stream it in using a stream analytics job. You can see that the structure of this .json file is a list of events, as specified by the square brackets, and every event is in the .json format. Embedded into every streaming event is a time field. This is the event time where the data originated at the sensor. We also have the sensor identifier, sensorC, E, and so on. This is a sensor that registered this event, and each event captures the temperature and humidity at that point in time. You can see that this is a list of streaming events. Let's switch back to the main page of our Azure portal. Here are all of the resources that we've used recently. Click through to the loonycsdstorage, the storage account for our blob storage container. Click through to blobs, and I'm going to upload this .json file to the container that I have here, the loony-csd-blob-container. Click on the Upload button. Find the file on your local machine and select it. Upload this file. This is the file that I'm going to stream in using a Stream Analytics job. Now that our file is present here within this blob container, use the left navigation pane, expand it, and select Create a resource. The resource that we are going to create here is a Stream Analytics job, and it's available under the Internet of Things category.

Stream Analytics on Azure is an on-demand real-time analytics service, and an analytics job uses an event processing engine designed to examine and transform high volumes of streaming data. It's in the IoT category because you would typically use Stream Analytics to stream in data from sensors. Let's configure our Stream Analytics job here. I'm going to call it loony-csd-streamjob. The subscription is pay-as-you-go, and the resource group is the loony-csd-rg. Click on Create. Azure will first validate the job, and once validation has been successful, it'll go ahead and deploy it. Once it's deployed, you can click on the Go to resource button to view your job. You now just have a job. It hasn't been executed yet. We'll run this job to read in data in a streaming format from our Azure blob storage container, and we'll visualize this data in Power BI. Container and input file is all set up. Let's move onto setting up a Power BI Pro account on the cloud that will allow us to view this data in real time. Go to powerbi.microsoft .com. Start Free, and once you get to the next page, you need to scroll down until you get to Power BI cloud collaboration and sharing. Unfortunately, this is a paid option, but you get a free 60-day trial. Power BI Desktop, which is completely for anyone to use, can't be used with Stream Analytics, unfortunately. So, Power BI Pro it is. Click on Try Free and let's get started. Use your email account and sign up for free. Here is my Dev User account. Specify a password. You'll get a verification code on your email account, which you'll then fill in here. And then click on Start. Using Power BI Pro, you can collaborate on your dashboards with your team. I'm going to skip sending invitations, and here I am. I'll be logged in just a few minutes. We are logged into our Power BI Pro account, and here within my workspace, you'll find the ability to create dashboards, reports, and workbooks.

Configuring Input and Output for the Stream Analytics Job

With Power BI set up to visualize our streaming data, let's head back to our Azure portal where we are on our loony-csd-streamjob. This is the Stream Analytics job that we had created in an earlier clip. We are now ready to configure the input for this job. This is the input from where streaming data will be ingested. Select the input option and let's begin configuring our stream input. Click on the plus button here to add a new stream input. If you're working on real world data, your input streaming data would be available either at an event hub or an IoT hub. For this demo, we'll stream in data from blob storage. That's where our InputStream.json file is located. This will bring up a pane that allows you to configure your input. The alias of the input is loonystreaminput, and we'll select the blob storage from our pay-as-you-go subscription. The storage account is the loonycsdstorage, and the storage account key has been auto-filled. Within this storage account, let's select the right blob container where that contains our data. This is the loony-csd-blob-container. The next step is to specify the event serialization format for your streaming data. Our input is in the form of a JSON file. We choose the JSON serialization format. The Stream Analytics job can also work with Avro and CSV files. Accept default values for all other options and hit Save. Azure will automatically test this connection to your streaming input. That was successful. Our connection to our streaming input has now been configured. Let's now configure the output for this Stream Analytics job. Select the output option and click on plus Add, and we'll output this streaming data to Power BI. You can see from the drop-down here that there a number of other options available to you as well. Table storage, you can put it into blob storage, you can use an Azure function to transform the data. There a number of services that you can choose to work with you streaming data. We'll choose to visualize it using Power BI. Let's give our output a name that's meaningful. I'll simply call it loonystreamoutput, and because this is a Power BI connection, we'll need to authorize that connection to load the Power BI workspace where data should be streamed. The streaming data will be available as a dataset in our Power BI workspace, and the name of this dataset will be loonystreamdataset. We'll stream our data into a table within this dataset called loonystreamtable. The next step is to authorize this connection to our Power BI Pro account that we just set up. Click on authorize, and you need to sign in. Dev.user @ loonycorn.com is our username. Specify the password, and you'll be authorized. I'll choose to stay signed in. My authorization is now complete. When you click on the Save button now, Azure will check that you are correctly authorized to connect to Power BI. The connection tests that are Azure performed to Power BI were successful. We are good to go. The actual processing performed by our Stream Analytics job can be specified using a function or a query. Click on Query here and we'll specify the query that we want to run to read in streaming data. This query here allows you to use a simple SQL-like format to specify data transformations if you want to. We can use as an input the loonystreaminput and as an output the loonystreamout. Both of these we've just configured. There are no other inputs and outputs possible. We haven't configured them. So, select the loonystreaminput as the source of our streaming data and the loonystreamoutput as our destination. Observe the structure of this query. I haven't specified any transformations. SELECT * into loonystreamoutput from loonystreaminput. We've intentionally kept our query very, very simple, but here is where you can specify grouping, aggregation, and window operations on your streaming data. Let's make sure that we are able to get data from our streaming input. Click on this three dot menu and choose Sample data from input. If you're successfully able to sample data from input, we know that our connection has been set up right. I will start the sampling right now, so I specify the current date and time and hit OK. This stream job will now start sampling the input from your streaming source. You can see that sampling the input was successful. Our connection has indeed been set up right. If you take a look at the events here, you'll see sample input succeeded.

Visualizing Streaming Data Using Power BI

We've sampled rows from our input. Our input configuration was fine. Let's test the Stream Analytics job as a whole. Click on the Test button here. Test will run this Stream Analytics job and pull in 1000 rows from your source. You can see that loonystreamoutput now has 1000 rows, and you can scroll down and take a look at your streaming events. Observe that the results here contain the fields from your original streaming events, the event time, the display name, the temperature, and the humidity. And in addition to the original fields, you can see that our job has added additional metadata to each record. Event processing time, the partition, blob name, and other details. At this point, I'm fairly happy with the configuration that I have for my Stream Analytics job. Click on Save and this will save the changes that we made to this particular job. Once the save is successful, we are ready to execute this job in real time. Let's head over to the overview page so that we can take a look at what exactly we are going to run and click on the Start button here. Now you can have this job start running right away, or you can schedule it for sometime in the future. I'll select the Custom option here and specify a few minutes from the time of this recording. I've scheduled this job to start running a few minutes from now. Now that the Stream Analytics job has started, let's start monitoring it. We'll up the pane here at the bottom. Here you can see the query that we are using to transform our streaming data. And right here on this page are some graphs that you can use to monitor the status of your streaming. We'll get a little alert when your job is up and running. Here it is. Wait for a little while until the graphs are populated. When I ran this Stream Analytics job, it took several minutes before I got actual visualizations on this dashboard. You can see from the graph here on the left that a few input events have streamed in and a few output events have been generated. You can click through and view this in more detail. This is great, but what we are really interested in doing is visualizing this information using Power BI. Switch over to the tab where you have Power BI Pro open and let's select and click through to my workspace. We haven't created any dashboards, reports, or workbooks yet, but if you select the Datasets tab, you'll find the loonystreamdataset. If you remember, this is the dataset that we had configured to be the output of our Stream Analytics job. This is pretty exciting, almost like magic. Observe this little chart icon here. This is what you'll select in order to create a dashboard to visualize your streaming data. And this takes us to a dashboard editor pane where the contents of our streaming data are available for us to visualize. Minimize the left navigation pane and let's create a chart or two. The first chart I'm going to select is a line chart. Select the line chart icon. This will automatically add it to your editor pane. Remember that this data is real time. As more data streams in, these charts will be updated. Select the humidity field and drag it onto the values section here within Power BI. In addition to humidity changes, I'll also view temperature changes on this line chart. Select the temperature field and drag it onto the values section as well. Our line chart will now display both humidity and temperature, and on the X axis, we'll select the time field. Select the time field and it'll be added to the X axis automatically, and here is our nice little line chart. You can select the knobs at the corner to expand this chart view and you can position it anywhere here on this edit pane. Let's add in one more chart so that our Power BI dashboard feels real. This time I'm going to select the bar chart option. We don't have very much interesting information in our streaming data, just temperature and humidity. I'm going to select the humidity field to be represented in the form of a bar. I'm going to select the temperature field as well. Select the temperature field off to the right. It'll be added to the values of this bar. And on the X axis, let's display the temperature and humidity by sensor. Select the display option, which gives us the sensor details. This bar chart automatically gives you a count of temperature and humidity events that we've received from each of our sensors. Remember that these are real time dashboards. As you're streaming data changes, these graphs will be updated.

Summary and Further Study

And with this we come to the very end of this module where we worked with streaming data. We first understood the differences between batch and stream processing. Batch works on bounded datasets, stream processing works on unbounded datasets. We then discussed how, if you need to perform aggregation operations on streams, we need to include all of the streaming entities that arrive within a window. When you're working with streaming data, the order in which the events occur is significant, and there are three types of time that you deal with, event time, ingestion time, and processing time. After we understood the basic concepts of stream processing, we moved onto hands-on demos where we saw how we could ingest streaming data with Stream Analytics on the Azure cloud platform. We then saw how we could visualize these streams in real time using Power BI. And this module brings us to the very end of this course. If you're interested in the Microsoft Azure cloud platform and interested in data processing, here are some other courses on Pluralsight that you might find interesting. Representing, Processing, and Preparing Data will talk about different data cleaning and preparation techniques using a variety of different technologies. Or if you're interested in performing univariate, bivariate, and multivariate analysis on your data and the base rule of conditional probability, this is the course for you, Summarizing Data and Deducing Probabilities. That's it from me here today. I hope you had fun with this course. Thank you for listening!

Course author

Janani Ravi

Janani has a Masters degree from Stanford and worked for 7+ years at Google. She was one of the original engineers on Google Docs and holds 4 patents for its real-time collaborative editing...

Course info

Level	Beginner
Rating	★★★★☆ (10)
My rating	★★★★★
Duration	3h 28m
Released	21 Jun 2019

Share course