ADX-Advanced-KQL

Materials for the ADX Advanced KQL Course

Read Me

This project contains demos and documentation.

The Demo folder contains the various code demos.

The Documentation folder includes detailed documentation for each of the demos. Please see the ReadMe.md file in the Documentation folder for more information.

Useful Links

This file contains useful links presented during the course.

Azure Data Explorer https://dataexplorer.azure.com/

Public Cluster https://help.kusto.windows.net/

Download the Kusto.Explorer desktop application https://docs.microsoft.com/en-us/azure/data-explorer/kusto/tools/kusto-explorer

Fixing the "Your admin has blocked this app due to security issues" error when installing the Kusto Explorer application

Creating ADX Clusters and Databases Using ARM templates https://docs.microsoft.com/en-us/azure/data-explorer/create-cluster-database-resource-manager

Manage Language Extensions in Your ADX Cluster https://docs.microsoft.com/en-us/azure/data-explorer/language-extensions

Azure Data Explorer Documentation https://docs.microsoft.com/en-us/azure/data-explorer/

Kusto Query Language (KQL) from Scratch https://app.pluralsight.com/library/courses/kusto-query-language-kql-from-scratch/table-of-contents

How to Start with Microsoft Azure Data Explorer https://www.pluralsight.com/courses/microsoft-azure-data-explorer-starting

VSCode Extensions

Suggested

- docs-markdown (Microsoft)
- Markdown Extension Pack
- Kuskus Kusto Extension Pack

Optional

- Python
- Bash Debug
- Better Comments
- Code Spell Checker
- Dummy Text Generator
- Excel Viewer
- Live Share
- PowerShell
- SandDance for VSCode
- PHP Debug
- PHP Intellisense
- SQL Server
- vscode-icons
- XML Tools
- YAML

Correcting the Installation Error

When installing the Kusto desktop application, you may get the error message:

Your administrator has blocked this application because it potentially poses a security risk to your computer.

This error happens because the ClickOnce trust prompt is disabled.

To fix, use **RegEdit**, and go to:

\HKEY_LOCAL_MACHINE\SOFTWARE\MICROSOFT.NETFramework\Security\TrustManager\PromptingLevel

Change the state of the required setting to Enabled.

Module 1 - Exploring User Analytics

Demo 1 - Sliding Window Counts

Overview

The goal of the `sliding_window_counts` plugin is to count user activity over time. The nifty thing about this plugin is it can count using a sliding window.

We should mention, a plugin is a library built by the Azure Data Explorer (ADX) team that has been added to the Kusto environment in order to extend the Kusto Query Language. The `sliding_window_counts` is one such plugin, we'll look at many others during this course.

Examining the Code

The first statement in our script creates the dataset we'll use for the demo. It has two columns, a user ID, in this case just a person's name, and a datetime value indicating the date for this row.

```
let T = datatable(UserId:string, Timestamp:datetime)
[
  // Bin 1: 06-01
  'Bob',      datetime(2020-06-01),
  'David',    datetime(2020-06-01),
  'David',    datetime(2020-06-01),
  'John',     datetime(2020-06-01),
  'Bob',      datetime(2020-06-01),
  // Bin 2: 06-02
  'Ananda',   datetime(2020-06-02),
  'Atul',     datetime(2020-06-02),
  'John',     datetime(2020-06-02),
  // Bin 3: 06-03
  'Ananda',   datetime(2020-06-03),
  'Atul',     datetime(2020-06-03),
  'Atul',     datetime(2020-06-03),
  'John',     datetime(2020-06-03),
  'Bob',      datetime(2020-06-03),
  // Bin 4: 06-04
  'Betsy',    datetime(2020-06-04),
  // Bin 5: 06-05
  'Bob',      datetime(2020-06-05),
];
```

Each row marks user activity, who did something and when it happened. Obviously in a real world situation there would be other columns, but these two are sufficient for this demo.

Here you can see both Bob and David were active twice on the first, as they both have two entries in the data. John only performed one activity on the first of June. You can see similar activity records for the remaining days.

We'll return to this dataset momentarily, for now let's look at the rest of the script.

In the next two lines, we simply create variables to hold a start date and an end date for the data range we want to analyze. Using variables to hold these, and other values, will make it easy to reuse the script to do analysis in the future.

```
let start = datetime(2020-06-01);
let end = datetime(2020-06-07);
```

In the next variable, we are setting the number of days for our sliding scale. Here, we are setting variable to indicate we want a window of 3 days to look back.

```
let lookbackWindow = 3d;
```

We now set our last variable, bin. The plugin will use the value in this variable to bin, or group our counts. Here we want to group the counts by day.

```
let bin = 1d;
```

If the data supported it, for example if the `timestamp` column also included the time of day, we could use other time values for `lookbackWindow` and `bin` variables, such as `1h` for hour.

Now we get to the fun part, the use of the `sliding_window_counts` plugin!

```
T | evaluate sliding_window_counts(UserId, Timestamp, start, end, lookbackWindow, bin)
```

We take our datatable, `T`, and pipe it into our `sliding_window_counts` plugin. We use the `evaluate` function to indicate we are calling a plugin.

The first value we pass into the plugin is the unique ID for a user. In this case it is the user name, found in the column `UserId`.

The next parameter represents the timeline data. Here it is the `Timestamp` column holding the date of our activity. The plugin uses this date for analysis and grouping.

The third parameter is the date to begin our analysis, represented by the `start` variable. In this simple example, the start date happens to coincide with the first row of data in our sample. It's much more likely though your data will span many days, months, or even years. This is why we need to let the plugin know where to start.

Likewise, we also need to let it know when to stop processing the data. We do so by passing in the `end` variable into the forth parameter.

A helpful hint, processing will go much faster if you limit the dataset to just the dates you need when you first read it from the source. Then you will only pipe in the limited dataset into the `sliding_window_counts` plugin.

The fifth parameter is the *look back window*, the number of days to use for our sliding scale. The plugin will look back the number of days in the `lookbackWindow` variable we are passing in.

For example, if the current date being processed is June 4th, and the look back window variable is set to three days, the plugin will aggregate the counts June 2nd, 3rd, and 4th.

The final parameter indicates how we want to bin, or group the data. In this case we pass in the variable `bin`, which is set to one day. Thus the results will aggregate totals per day.

Analyzing the Output

Let's look at the result of the query.

Timestamp	Count	Dcount
2020-06-01 00:00:00.0000000	5	3
2020-06-02 00:00:00.0000000	8	5
2020-06-03 00:00:00.0000000	13	5
2020-06-04 00:00:00.0000000	9	5
2020-06-05 00:00:00.0000000	7	5
2020-06-06 00:00:00.0000000	2	2
2020-06-07 00:00:00.0000000	1	1

The resultant data table contains three columns. The first is the `timestamp` which is how our results are grouped.

The next column is the `count`. If you look back at the input data table, it contained five rows on June 1st. This went into the `count` value for this first row in the output.

The final column is `dcount`, short for distinct count. It counts based on the number of unique values in the ID column, in this case the `UserId`.

In the input data table, John appears once on June 1st, Bob and David twice. However, these are three distinct users, thus the `dcount` column holds the value 3.

Let's shift down, so we can get an idea of the sliding window in action. Specifically look at the output for June 4th, where we have a count of 9.

Looking back at the input data table, our sliding window goes back three days. Thus for June 4th, the three day window will cover the dates of June 2nd, 3rd, and 4th.

On June 2nd, there are 3 rows. On June 3rd, we have 5 rows. Finally we have only 1 row on June 4th, which gives a total of 9 rows. This matches the value in the `Count` column of our output for June 4th.

In that same period, we had five users appear in the rows. Ananda, Atul, John, Bob, and Betsy. This corresponds to the output's `Dcount` column.

Summary

While simple, this example clearly illustrates how the `sliding_window_counts` plugin works in KQL.

Module 1 - Exploring User Analytics

Demo 2 - Active User Counts

Overview

The goal of the active user counts plugin is to help identify your "fans", in other words people who regularly use the product or service you are logging activity for. The data being analyzed must have an identifier to uniquely identify the user, as well as a date/time the activity was recorded on.

There are a few key factors to determining if someone is active within the time period. First is the period. If, for example, it is set to day, we will see if a user is active within that day. We can also use other units of time such as hours, minutes, weeks, months, and so on.

The next factor is the look back window, it is an amount of time that uses the same time unit that was used in the period. For example, if our period was set to day, the look back is how many days we want to look back from the date being analyzed. If this looks familiar it should, in the previous demo the sliding window counts used the concept as well.

Finally we need to tell the plugin how many periods a user must appear in to be counted as a "fan". It's starting to sound a bit complex, but perhaps this simple example will illustrate the rule.

If we set the period to 1 day, look back window to 5 days, and active periods to 3, it means that a user must be active 3 days out of the 5 days. Note that the user must appear on 3 separate days. If they were to appear 3 times but all on the same day, they won't be counted.

Examining the Code - Example 1 - The Basic Query

We start the query by creating a datatable with the data to examine. Note comments were added to make it clear where the data falls into the bins we'll be using.

```
let T =  datatable(User:string, Timestamp:datetime)
[
    // Pre-start date
    "Bob",      datetime(2020-05-29),
    "Bob",      datetime(2020-05-30),
    // Bin 1: 6-01
    "Bob",      datetime(2020-06-01),
    "Jim",      datetime(2020-06-02),
    // Bin 2: 6-08
    "Bob",      datetime(2020-06-08),
    "Bob",      datetime(2020-06-09),
    "Jim",      datetime(2020-06-10),
    "Bob",      datetime(2020-06-11),
    "Jim",      datetime(2020-06-14),
    // Bin 3: 6-15
    "Bob",      datetime(2020-06-21),
    "Jim",      datetime(2020-06-21),
    // Bin 4: 6-22
    "Jim",      datetime(2020-06-22),
    "Bob",      datetime(2020-06-22),
    "Jim",      datetime(2020-06-23),
    "Bob",      datetime(2020-06-24),
    // Bin 5: 6-29
    "Bob",      datetime(2020-06-24)
];
```

We'll return back to the data momentarily, for now let's look at the next part of the query in which we set a few additional variables.

```
let Start = datetime(2020-06-01);
let End = datetime(2020-06-30);
let Period = 1d;
let LookbackWindow = 5d;
let ActivePeriods = 3;
let Bin = 7d;
```

The `Start` and `End` indicate the range we want to analyze within our data.

The `LookbackWindow` indicates how many days backwards the plugin should look for activity from the same user. Keep in mind that the lookback can go previous to the start date. If we refer back to the dataset, you will note Bob was active on June 1st. When the `active_user_counts` plugin analyzes that row of data, it will go back five days and look for activity for Bob in the range of May 27 to June 1st.

The next variable is the `Period`. Here it is set to 1d or 1 day. It doesn't matter how many times a user was active in that time period, Bob could have been active 1 time or 20 on a single day, it would still only be counted once when being evaluated for activity.

`ActivePeriods`, the next value, says how many time periods a user must be in to be considered active. In this example, a user must show up at least 3 days out of the last 5 days (the look back window), to be counted as active.

The final variable sets how we want to bin, or group our results. Here we set `bin` to 7 days. Since our start date is June 1, our bins will be June 1, 8, 15, 22, and 29. The 29th is the final bin as anything beyond that would exceed the end date set in the `End` variable.

Now we get to the core of this demo, the active_users_count plugin.

```
T | evaluate active_users_count(User, Timestamp, Start, End, LookbackWindow, Period, ActivePeriods, Bin)
```

Here we take our datatable, contained in the variable T, and pipe it into the evaluate function. Evaluate is needed to let Kusto know this is a plugin we are calling and not a native KQL function.

The first parameter we pass to the `active_users_count` plugin is the unique identifier that marks a user. Here we use the User column name from our datatable.

Next we indicate what column from the datatable to use as the time. Here it comes from the appropriately named Timestamp column in the source datatable.

The remaining parameters correspond to the variables we created, since their purpose was explained in the variables section we won't reiterate here.

Analyzing the Output

Let's run the query, and take a look at it's output.

Timestamp	dcount
2020-06-01 00:00:00.0000000	1
2020-06-08 00:00:00.0000000	1
2020-06-22 00:00:00.0000000	2

To see where these results came from, let's refer back to the input datatable.

```
// Pre-start date
"Bob",      datetime(2020-05-29),
"Bob",      datetime(2020-05-30),
// Bin 1: 6-01
"Bob",      datetime(2020-06-01),
"Jim",      datetime(2020-06-02),
```

The process begins on June 1st, and will process all dates in our first bin. The bin period was set for 7 days, so the date range for bin 1 is June 1 to June 7.

The first record the plugin will examine from the first bin is Bob, on 2020-06-01. It will then look back 5 days into the data, even if those dates occur prior to the June 1 start date. Thus the two records for Bob, the 29th and 30th, will be evaluated.

Since the active periods was set to 3, and Bob has been active three times in the five day window (May 27 to June 1), he is counted.

The next row that is read is for Jim on June 2. Looking back, Jim has no other records during the bin or in the five days before it, so he won't be included in the results.

This yields the first row in our output. It shows the starting date for the bin, 2020-06-01, and the distinct count. Please note this is a *distinct* count.

Bob could also have been active on June 3rd, 4th, and 5th. However because he was already showing up in the results for his activity of May 29 to June 1, he would not be counted a second time.

Let's look now at bin 2, for June 8. In the output it shows 1 distinct active user.

```
// Bin 2: 6-08
"Bob",      datetime(2020-06-08),
"Bob",      datetime(2020-06-09),
"Jim",      datetime(2020-06-10),
"Bob",      datetime(2020-06-11),
"Jim",      datetime(2020-06-14),
```

This uses the same logic pattern already described, and Bob was the only user who qualified under those rules. Hence we have the 2020-06-08 bin in the output with a distinct count of 1.

Next up is bin 3, for June 15. As you can it only has two rows. Neither has any data that occur in our five day lookback window.

```
// Bin 3: 6-15
"Bob",      datetime(2020-06-21),
"Jim",      datetime(2020-06-21),
```

Now if you look at the output you'll see.... oh, wait, actually you *won't* see. You won't see a row for this bin date, that is.

This is a very important detail to note. If the active_users_count plugin does not find any distinct users, in other words the distinct count is zero, then it does not return any rows.

There may be instances where you want to note that a particular bin date had zero active users, and we'll see how to solve that dilemma in a moment.

Before we get to that solution, let's examine bin 4.

```
// Bin 3: 6-15
"Bob",      datetime(2020-06-21),
"Jim",      datetime(2020-06-21),
// Bin 4: 6-22
"Jim",      datetime(2020-06-22),
"Bob",      datetime(2020-06-22),
"Jim",      datetime(2020-06-23),
"Bob",      datetime(2020-06-24),
```

You may think a copy/paste error occurred as bin 3 is also appearing, but that was done on purpose. Both Bob an Jim have two records each in the fourth bin. However, the two rows in bin 3 fall into the five day lookback window. As a result, they meet the criteria for three active entries in the five day period, and are included in the output for June 22nd.

Finally, we have data for the final bin.

```
// Bin 5: 6-29
"Bob",      datetime(2020-06-29)
```

As there is only one row here, and the data for it does not meet our criteria for inclusion no row appears for this bin in the output.

Examining the Code - Example 2 - Adding in Missing Bin Dates

In the previous section we mentioned how to fill in the missing bin dates into our output. The first step is to add two lines of code to the bottom of our query.

```
| join kind=rightouter (print Timestamp = range (Start, End, Bin)
| mv-expand Timestamp to typeof(datetime)) on Timestamp
```

The first row uses the range command to create a small table of dates for the bin dates and joins it to the output, the second line then converts the joined output to be all datetimes. Let's look at the output.

Timestamp	dcount	Timestamp1
		2020-06-15 00:00:00.0000000
		2020-06-29 00:00:00.0000000
2020-06-01 00:00:00.0000000	1	2020-06-01 00:00:00.0000000
2020-06-08 00:00:00.0000000	1	2020-06-08 00:00:00.0000000
2020-06-22 00:00:00.0000000	2	2020-06-22 00:00:00.0000000

As you can see, it added a new column `Timestamp1` to the output. For the two rows that represent the missing bins, it has placed nulls (where you see the empty columns) for the original `Timestamp` and `dcount` columns.

Progress, but not very appealing to the end user. But we can tidy it up easily, and will do so in the next section.

Examining the Code - Example 3 - Cleaning up the Output

Here is the final line in our query that will clean up the output.

```
| project Timestamp=coalesce(Timestamp, Timestamp1), dcount=coalesce(dcount, 0)
```

In it, we are using the `project` command to create two new columns in our output stream. Both of these use the `coalesce` function. Coalesce works by taking the first value being passed in, and comparing it to null. If it is not null, it uses the value in that first parameter. If on the other hand it is null, then it moves onto the second parameter and repeats.

In our code we only need to use two parameters with coalesce, but you can actually have a long list. It will just repeat the logic of null checking as described above until it either finds a non-null value or runs out of parameters.

After checking the timestamp, it then repeats with the dcount, either returning the dcount value from the active_users_count plugin or the value of 0.

Let's see how this looks in our output.

Timestamp	dcount
2020-06-01 00:00:00.0000000	1
2020-06-08 00:00:00.0000000	1
2020-06-15 00:00:00.0000000	0
2020-06-22 00:00:00.0000000	2
2020-06-29 00:00:00.0000000	0

Much better! As you can see, our original 3 rows for June 1, 8, and 22 are present. In addition, we now have rows for the bins of June 15 and 29, where the dcount was 0.

Summary

Although it can seem a bit confusing at first, once you understand the logic behind the `active_users_count` plugin it becomes straight forward. In addition, despite the name it can be used for other applications beyond just users.

For example instead of counting the unique users, you could count the number of frequently occurring, unique error messages that occur during a period.

Now that you understand how it works, you can use it for many applications in your environment.

Module 1 - Exploring User Analytics

Demo 3 - Activity Counts Metrics

Overview

The `activity_counts_metrics` plugin provides both a count and a distinct count for activity in a time period. In addition, it provides two other key metrics. One is an aggregated distinct count for the current time period plus all previous time periods. The second is a new distinct count. This has a count of activities that are new in the time period being analyzed, in other words rows that have not appeared in previous time periods.

Examining the Code - Example 1 - Basic Output

We start our query with our data source, as we did in previous demos. We create a datatable with user IDs and dates.

```
let T = datatable(UserId:string, Timestamp:datetime)
[
  // June 1
  'Bob',   datetime(2020-06-01),
  'John',  datetime(2020-06-01),
  // June 2
  'Cindy', datetime(2020-06-02),
  'John',  datetime(2020-06-02),
  'Ted',   datetime(2020-06-02),
  // June 3
  'Bob',   datetime(2020-06-03),
  'John',  datetime(2020-06-03),
  'Todd',  datetime(2020-06-03),
  'Todd',  datetime(2020-06-03),
  'Sam',   datetime(2020-06-03),
  // June 5
  'Sam',   datetime(2020-06-05),
];
```

Next, we setup a few basic variables.

```
let start=datetime(2020-06-01);
let end=datetime(2020-06-05);
let window=1d;
```

As with previous demos, `start` and `end` are used to mark the date range we want to analyze.

The `window` variable is how we want to bin, or group our results. Here we are using a single day, but other values such as weeks, months, minutes, hours, and more are valid.

In the final line of the query we call the `activity_counts_metrics` plugin.

```
T | evaluate activity_counts_metrics(UserId, Timestamp, start, end, window)
```

As in other demos we use the datatable being held in the variable T and pipe it through the `evaluate`, needed as we are calling a plugin. Then is the plugin itself.

In the first parameter we pass in the column with the unique key for this datatable. Here we are using a user ID, but it could be anything you want to analyze. Product ID, error message, and more.

The last three are straight forward. The start and end range for the dates to analyze, and the way we want to group the output, in this case our window of 1 day.

Analyzing the Output

Before we look at the output, for easy reference, here is our data nicely formatted. To make it easier to read, repeating dates are suppressed.

Timestamp	UserID
2020-06-01	Bob
	John
2020-06-02	Cindy
	John
	Ted
2020-06-03	Bob
	John
	Todd
	Todd
	Sam
2020-06-05	Sam

Running our query, let's look at the output.

Timestamp	count	dcount	new_dcount	aggregated_dcount
2020-06-01 00:00:00.0000000	2	2	2	2
2020-06-02 00:00:00.0000000	3	3	2	4
2020-06-03 00:00:00.0000000	5	4	2	6
2020-06-05 00:00:00.0000000	1	1	0	6

On the first row of output, we have a count of 2, representing the two rows in the data source for June 1. We also have two distinct users for this day, Bob and John, so our distinct count is also 2.

Because there are no previous rows, the new distinct count and aggregated distinct count are both 2 as well.

In the second row of output, for June 2, we start to see how this plugin works. Count is straight forward, there are 3 rows on this date so the count is 3.

Additionally we have three distinct users on this day, Cindy John and Ted, so the distinct count is 3.

For the new distinct count, we have 2, Cindy and Ted as they did not appear in any previous day. John appeared on the 1st, so he is not counted in this value.

The aggregated distinct count represents a grand total of unique users for this and previous days, giving us a value of 4. Bob and John on the 1st, Cindy and Ted on the 2nd.

On June 3rd we have five rows, so the count is 5. In those 5 rows we only have 4 unique users, as Todd had two activities logged on that date, so our distinct count is 4.

Of these users, Todd and Sam are new, the other users appeared on previous dates. Thus the new distinct count is again 2.

Finally we have the aggregated distinct count. For the third we are adding in Todd and Sam to the activities of Bob, Cindy, John, and Ted from previous dates, giving us a total of 6.

Note that we don't have a row for June 4th, this is because we have no data on that date. Like the previous demo, rows with 0 values are not returned. We'll see how to handle that in a moment, but for now let's look at that last row of data.

Sam appears on June 5th, and is the only user on that date. So, both count and dcount are 1.

As Sam appeared on previous days there are no new distinct users on this date, so new dcount is going to be 0. Likewise, as there are no new users being active, the aggregated distinct count will remain at 6.

If this is the output you want, and you aren't worried about the missing row for June 4th, great! You're done. If however you want to include that row of zero data in your output, read on.

Examining the Code - Example 2 - Adding Missing Days

Now let's add a few lines to the query so we can let users know we had no activity on a missing date.

```
| join kind=rightouter (print Timestamp = range (start, end, window)
| mv-expand Timestamp to typeof(datetime)) on Timestamp
| project Timestamp=coalesce(Timestamp, Timestamp1)
        , count=coalesce(['count'], 0)
        , dcount=coalesce(dcount, 0)
        , new_dcount=coalesce(new_dcount, 0)
        , aggregated_dcount=coalesce(aggregated_dcount, 0)
```

The first two lines are the same as our previous demo. They create a small table, with one row for each date in our start to end date range. This is then joined to the output of the `activity_counts_metrics` plugin.

The `project` command, beginning in line 3, is all one command. We just wrap it onto new lines to make it easier to read. For each column, we are again using coalesce to compare the output from the `activity_counts_metrics` plugin to null, and if null fill in with either the timestamp generated using the range, or a value of 0.

Note one minor formatting convention within the statement `coalesce(['count'], 0)`. Count is the name of an operator in KQL. In order to let our query know we want to use this as a column name, and not call the count operator, we surround it with ['']. This then gives us our output to include our missing date of June 4th.

Timestamp	count	dcount	new_dcount	aggregated_dcount
2020-06-01 00:00:00.0000000	2	2	2	2
2020-06-02 00:00:00.0000000	3	3	2	4
2020-06-03 00:00:00.0000000	5	4	2	6
2020-06-04 00:00:00.0000000	0	0	0	0
2020-06-05 00:00:00.0000000	1	1	0	6

Summary

The `activity_counts_metrics` plugin provides a great way to calculate not just counts for individual days, but to keep a running aggregated distinct count across a range of dates.

Module 1 - Exploring User Analytics

Demo 4 - Activity Metrics

Overview

The plugin `activity_metrics` will return valuable information about user activity for a period as compared to the previous time period. The first two are basic distinct counts. One is simply the number of distinct users that appear in the time period being examined. The second is the number of new users, in other words users that appear in the time period under examination who were not found in the previous period.

The next two are ratios in the value of 0 to 1. The retention rate indicates how many users from the last period returned for this period. For example if 2 of 3 users in the previous period returned, that ratio would be 2/3 or 0.66666667 (i.e. 66.67%).

Churn rate is a ratio of how many people are lost compared to the previous time period, and is calculated similarly. If 1 out of 3 users in the previous period did not return for the current period, the rate is 1/3 or 0.33333333 (or 33.33%).

Examining the Code - Demo 1 - Basic Query

As usual with a query you should start with data. Here we generate a datatable and store in a variable, `T`.

```
let T =  datatable(User:string, Timestamp:datetime)
[
    // Bin 1: 6-01
    "Bob",      datetime(2020-06-01),
    "Jim",      datetime(2020-06-02),
    "Jim",      datetime(2020-06-03),
    // Bin 2: 6-08
    "Bob",      datetime(2020-06-08),
    "Sue",      datetime(2020-06-09),
    "Jim",      datetime(2020-06-10),
    // Bin 3: 6-15
    "Bob",      datetime(2020-06-21),
    "Jim",      datetime(2020-06-21),
    // Bin 4: 6-22
    "Ted",      datetime(2020-06-22),
    "Sue",      datetime(2020-06-22),
    "Bob",      datetime(2020-06-23),
    "Hal",      datetime(2020-06-24),
    // Bin 5: 6-29
    "Bob",      datetime(2020-06-29)
];
```

Next, we will create a few simple variables to hold the start and end dates for the range of data we want to examine. In addition we will declare a bin value, in other words how to group our data. This is set to 7d for 7 days.

```
let Start = datetime(2020-06-01);
let End = datetime(2020-06-30);
let bin = 7d;
```

Finally we call the `activity_metrics` plugin.

```
T | evaluate activity_metrics(User, Timestamp, Start, End, bin)
```

The values being passed in are pretty obvious. The first is the value we can use for a key to uniquely identify a row of data, here we use the `User` column from the datatable. Next is the datetime data column to analyze on, here called `Timestamp`.

Next up is the start and end date to indicate the range to use for analysis. Finally is the value to be used to group our data, `bin`.

Analyzing the Output - Example 1

Timestamp	dcount_values	dcount_newvalues	retention_rate	churn_rate
2020-06-01 00:00:00.0000000	2	2	1	0
2020-06-08 00:00:00.0000000	3	1	1	0
2020-06-15 00:00:00.0000000	2	0	0.666666666666667	0.333333333333333
2020-06-22 00:00:00.0000000	4	3	0.5	0.5
2020-06-29 00:00:00.0000000	1	0	0.25	0.75

The first output row contains the data for bin 1, which began on June 1st. Looking at the data we have one entry for Bob, and two for Jim. Remember the d in dcount stands for distinct, so the first column has a value of 2.

As this is the first row, we have no previous row to compare to, so both are counted as new distinct users. Likewise, with no previous data the retention rate is set to 1, or 100%, and the churn to 0.

In the next bin of June 8, Bob and Jim return and are joined by a new user, Sue giving a distinct count of 3. We have one new user, Sue, so dcount_newvalues has a value of 1.

As both Bob and Jim returned, we have 2 of 2 coming back so the retention rate is 2/2 or 1 (100%). Likewise, as we didn't lose anyone the churn rate is 0.

Now we move to the data for June 15th. Once again Bob and Jim are back, but Sue did not return. By now you can guess the dcount_values will be 2, and with no new users the dcount_newvalues is 0.

For the retention rate, 2 of our 3 users came back, divided out becomes 0.666666667 (or 66.67%).

In the previous time period of June 8, we had 3 users. One of those users, Sue, did not return. 1/3 is 0.33333333 or 33.33%.

Looking at the fourth bin, for June 22nd, we have four users. Three of these are brand new. Of the two users from the last time period, Jim and Bob, only Bob returned. 1 returning of 2 yields a retention of 0.5. 1 leaving of the 2 also yields 0.5 for the churn rate.

Moving to the last bin, for June 29, we had one user, with no new ones. As only one of the four returned, the retention was 0.25 or 25%. We lost three of the four, making our churn rate 0.75.

Examining the Code - Demo 2 - Formatting the Output

Our previous work is perfect for piping into the next part of the query, or exporting for further analysis. However, it is not very easy to read. Should you wish, you can format the output to make it easier for people to refer to.

We'll do so via the `project` command. Add these few lines to the bottom of the query from example 1.

```
| project TimePeriod=format_datetime(Timestamp, 'MM/dd/yyyy')
        , ActiveUsers = dcount_values
        , NewUsers = dcount_newvalues
        , RetentionPercent = round((retention_rate * 100), 2)
        , ChurnPercent = round((churn_rate * 100), 2)
```

Using `project` we can take our original set of output columns from the query and reformat them. The KQL function `format_datetime` is used to format out timestamp to a readable value. Note the use of capital M's for the month.

We opted not to format the active and new user distinct counts, although we do rename them for the output. To make the rates easy to read, we start by multiplying both by 100 in order to convert them to percentages. The `round` function is then use to round them off to just two decimal places. Finally, they are renamed to add the word *Percent* to the end to make it clear these are now percentage values.

TimePeriod	ActiveUsers	NewUsers	RetentionPercent	ChurnPercent
06/08/2020	3	1	100	0
06/15/2020	2	0	66.67	33.33
06/22/2020	4	3	50	50
06/29/2020	1	0	25	75
06/01/2020	2	2	100	0

Summary

As you can see from this demo, the `activity_metrics` plugin makes it easy to create row over row comparisons for new user counts, as well as retention / churn values.

Module 1 - Exploring User Analytics

Demo 5 - Activity Engagement

Overview

Using the `activity_engagement` plugin, you can determine the number of distinct active users on a particular date. In addition, you can determine the number of active users in a sliding window for the specified time period prior to that date. Finally, it will produce a ratio of distinct active users on a date compared to the number in the sliding window.

Let's look at a simple example. On June 7 your data shows 2 active users. You specified a sliding window of 7 days, so the plugin will aggregate the total number of distinct users from June 1 to June 7. For our example, let's say the total is 10. On June 7th then, 2 active users out of 10 total yields a ratio of 0.2, or 20%.

Examining the Code - Example 1 - Basic Query

We'll start by creating a datatable, and storing in the variable `T`.

```
let T =  datatable(User:string, Timestamp:datetime)
[
    "Bob",      datetime(2020-06-01),
    "Jim",      datetime(2020-06-02),
    "Jim",      datetime(2020-06-03),
    "Ted",      datetime(2020-06-03),
    "Sue",      datetime(2020-06-04),
    "Bob",      datetime(2020-06-04),
    "Jim",      datetime(2020-06-04),
    "Ann",      datetime(2020-06-05),
    "Mac",      datetime(2020-06-06),
    "Lee",      datetime(2020-06-07),
    "Bob",      datetime(2020-06-08),
    "Sue",      datetime(2020-06-09),
    "Jim",      datetime(2020-06-10),
    "Bob",      datetime(2020-06-11),
    "Jim",      datetime(2020-06-11),
    "Ted",      datetime(2020-06-12),
    "Sue",      datetime(2020-06-12),
    "Bob",      datetime(2020-06-12),
    "Hal",      datetime(2020-06-12),
    "Bob",      datetime(2020-06-12)
];
```

Next, we'll set a few variables. First, we'll store the start and end dates to examine our data for.

```
let Start = datetime(2020-06-01);
let End = datetime(2020-06-30);
```

Next, we set the inner and outer activity windows.

```
let InnerActivityWindow = 1d;
let OuterActivityWindow = 7d;
```

The inner activity window indicates how we should group data for purposes of comparison. Here we use 1d (1 day), thus activity will be aggregated on a daily basis.

The second variable, OuterActivityWindow, is the length of time to use for our sliding scale. Here we use 7d (7 days). When we process a record for June 7th, for example, the plugin will look back 7 days and total the distinct users for the time range of June 1 to June 7.

Finally we are ready to call the plugin.

```
T | evaluate activity_engagement(User, Timestamp, Start, End, InnerActivityWindow, OuterActivityWindow)
```

As with calling other plugins, we pass our datatable into the evaluate, which will let KQL know this is a plugin.

For the first parameter we pass in the User column name, which will be used as the key for determining distinctness. Next is the Timestamp, used for the date evaluation.

The start and end parameters are pretty obvious. Finally we pass in the value to use for our inner bin value, and the value for the length of time to look back.

Analyzing the Output

Running the query using the above data returns the following data. Note that since we selected 1d for our InnerActivityWindow, the data is broken down at one row per day.

Timestamp	dcount_activities_inner	dcount_activities_outer	activity_ratio
2020-06-01 00:00:00.0000000	1	1	1
2020-06-02 00:00:00.0000000	1	2	0.5
2020-06-03 00:00:00.0000000	2	3	0.666666666666667
2020-06-04 00:00:00.0000000	3	4	0.75
2020-06-05 00:00:00.0000000	1	5	0.2
2020-06-06 00:00:00.0000000	1	6	0.166666666666667
2020-06-07 00:00:00.0000000	1	7	0.142857142857143
2020-06-08 00:00:00.0000000	1	7	0.142857142857143
2020-06-09 00:00:00.0000000	1	7	0.142857142857143
2020-06-10 00:00:00.0000000	1	6	0.166666666666667
2020-06-11 00:00:00.0000000	2	6	0.333333333333333

Timestamp	dcount_activities_inner	dcount_activities_outer	activity_ratio
2020-06-12 00:00:00.0000000	4	7	0.571428571428571

Let's look at the data in the output for June 7. It shows 1 distinct user on this date. Referring to our input data, we see that Lee was indeed active on the 7th, and the only person to have data recorded on that date.

Now we need to look back 7 days, the length of time indicated in the OuterActivityWindow variable. During that time, June 1 to June 7, the following users were active: Bob, Jim, Ted, Sue, Ann, Mac, and Lee. Seven users, which is the value in the dcount_activities_outer column of the output.

Finally we need to calculate the ratio of distinct users for a day to the distinct user count for the sliding window. 1 divided by 7 is 0.142857142857143, or roughly 14%.

This same logic applies to all rows in the output. Note that if there is no data previous to the start date, it uses a value of 0. Thus you may wish to include a few extra rows of data for dates prior to the start date in order to get accurate data for the first few rows, as we did in the `active_users_count` demo.

Examining the Code - Example 2 - Formatting the Output

As with the previous demo, let's take a moment to format the output. Add the following lines to the bottom of the query.

```
| project TimePeriod=format_datetime(Timestamp, 'yyyy-MM-dd')
        , ActiveUsersToday = dcount_activities_inner
        , ActiveUsersInTimePeriod = dcount_activities_outer
        , ActivityPercent = round((activity_ratio * 100), 2)
| order by TimePeriod asc
```

We'll reformat the time, this time we'll use the yyyy-MM-dd format. The two counts we simply use project to rename. We then take the activity_ratio, multiply by 100 to turn it into a percentage, round it to just two decimal places, and give it a nice name.

Finally we pipe it to an order command in order to ensure the output is sorted in the order we need.

TimePeriod	ActiveUsersToday	ActiveUsersInTimePeriod	ActivityPercent
2020-06-01	1	1	100
2020-06-02	1	2	50
2020-06-03	2	3	66.67
2020-06-04	3	4	75
2020-06-05	1	5	20
2020-06-06	1	6	16.67
2020-06-07	1	7	14.29
2020-06-08	1	7	14.29
2020-06-09	1	7	14.29
2020-06-10	1	6	16.67
2020-06-11	2	6	33.33
2020-06-12	4	7	57.14

Summary

The 'activity_engagement` plugin provides a simple way to compare activity on a particular slice of time to a sliding window of periods up to and including the time being examined. In this demo, we compared activity for a day to the window of 7 days.

Module 2 - Geoclustering

Demo 1 - Nearby Events

1.1 Geo Point in Circle - Overview

In this demo, we will look at two functions that calculate proximity for geographic data. The first of these is geo_point_in_circle. As its name implies, it will return data whose coordinates fall within a specified radius of a center point.

1.1.1 Examining the Code

We'll begin this demo with a simple example. We'll start by declaring a datatable and placing it into a variable, T. It will hold three columns, a longitude, latitude, and a place name.

```
let T = datatable(longitude:real, latitude:real, place:string)
[
    real(-122.317404), 47.609119, 'Seattle',                  // In circle
    real(-123.497688), 47.458098, 'Olympic National Forest',  // In exterior of circle
    real(-122.201741), 47.677084, 'Kirkland',                 // In circle
    real(-122.443663), 47.247092, 'Tacoma',                   // In exterior of circle
    real(-122.121975), 47.671345, 'Redmond',                  // In circle
];
```

Next, we will declare three variables. The first is the radius, in meters, for our circle. Next we declare the longitude and latitude for the center point of our circle.

```
let radius = 18000;
let centerLong = -122.317404;
let centerLat = 47.609119;
```

Finally, we will take our data table, stored in the variable T, and pass it into the `geo_point_in_circle` function.

```
T | where geo_point_in_circle(longitude, latitude, centerLong, centerLat, radius)
  | project place
```

In the first two parameters, we pass in the names of the columns from the source data table, in this case T, that correspond to the longitude and latitude. In the next two positions we pass in the long/lat for the center point of our circle. Finally, we pass in the radius for our circle.

This acts as a filter, only rows of data that are inside the radius of our circle will get passed through. As a final step we reduce our output to just contain the names of the places within our circle.

1.1.2 Analyzing the Output

Running the query returns a simple output.

place
Seattle
Kirkland
Redmond

In this example, the `geo_point_in_circle` function was used to filter our data down to just the rows that fall inside our circle.

Let's look at something a little more advanced for this next example.

1.2.1 Examining the Code

For this example we will draw data from the StormEvents table, part of the Azure Data Explorer examples.

This table includes events, such as thunder storms, hail, wind and other weather related events, along with the longitude and latitude where they occurred. We'll analyze events from this table, this time plotting the results to a map.

```
let radius = 10000;
let centerLong = -80.6048;
let centerLat = 28.0393;
StormEvents
  | where isnotempty( BeginLat) and isnotempty( BeginLon)
  | project BeginLon, BeginLat, EventType
  | where geo_point_in_circle(BeginLon, BeginLat, centerLong, centerLat, radius)
  | render scatterchart with (kind = map)
```

As in the previous example, we have variables setup to hold our radius, in this case 10,000 meters, along with the longitude and latitude for the center of our circle.

The storm events table stores the location an event was first spotted in the BeginLat and BeginLon column names. It's important we have that data for the `geo_point_in_circle` function to work. Since that data may not always be present, we pipe our storm events table through a `where` clause to remove any rows where either of these pieces of data are missing.

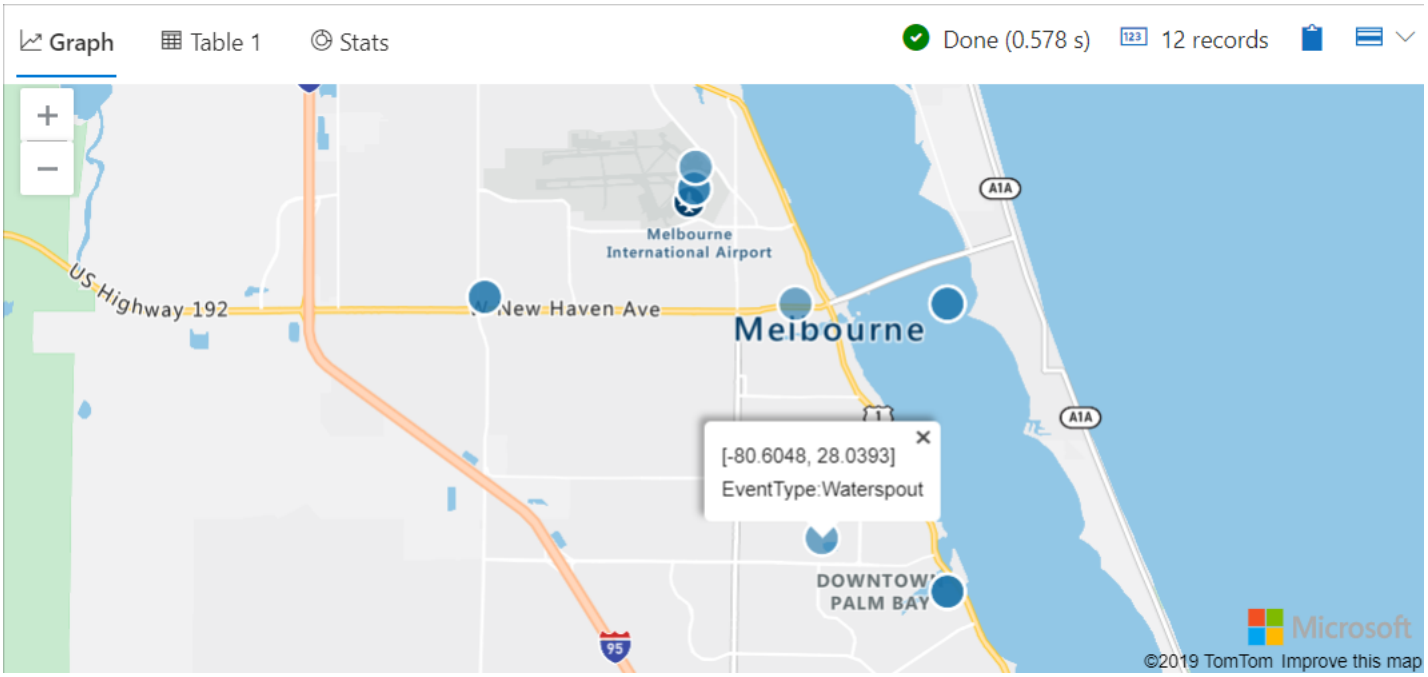
A key to dealing efficiently with data inside Azure Data Explorer is to trim down that data as much as possible. As such, in the `project` line we'll strip the data to just the three columns we'll need.

Now we pass the data into the `geo_point_in_circle` function, using it to filter out events not inside our circle.

As a last step, we then render the output as a scatterchart, adding the `kind = map` to indicate we want the scatter chart to be overlaid upon a map.

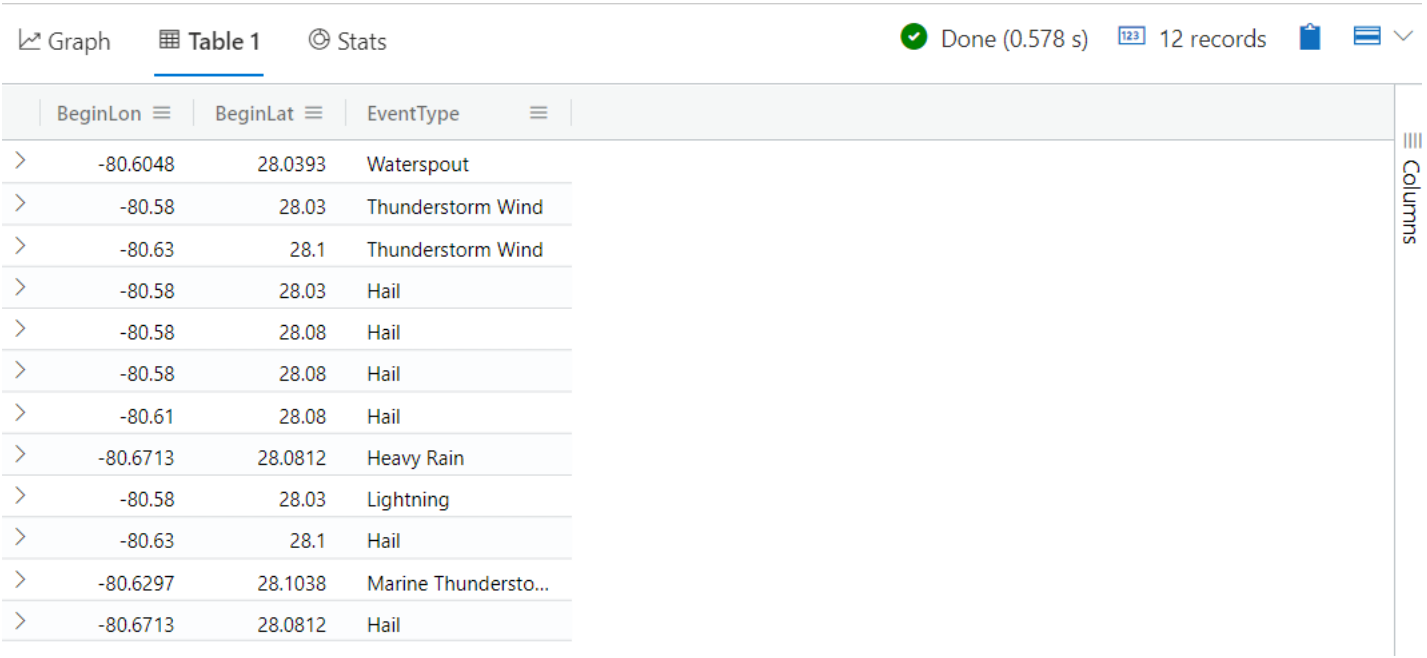
1.2.2 Analyzing the Output

Looking at the output, we see a nice map, with dots where specific events occurred within our circle.



Note that the center point for the circle was also an event, a waterspout. Clicking on it revealed the exact coordinates, along with the name of the event.

If you wish to see the output in text, simply click on the Table tab above the map.



BeginLon	BeginLat	EventType
-80.6048	28.0393	Waterspout
-80.58	28.03	Thunderstorm Wind
-80.63	28.1	Thunderstorm Wind
-80.58	28.03	Hail
-80.58	28.08	Hail
-80.58	28.08	Hail
-80.61	28.08	Hail
-80.6713	28.0812	Heavy Rain
-80.58	28.03	Lightning
-80.63	28.1	Hail
-80.6297	28.1038	Marine Thundersto...
-80.6713	28.0812	Hail

1.3 Geo Distance Point to Line - Overview

In our last example, we used geo_point_in_circle to filter our data to only those points that existed within a circle. What if we didn't have a nice neat circle though?

What if we needed to find storm events that were located in close proximity to, let's say, a major highway? Well that's where the geo_distance_point_to_line comes in.

1.3.1 Examining the Code

```
let distanceFromLine = 500;  // 500 meters
let geoLine = dynamic( { "type":"LineString"
                       , "coordinates":[ [-81.76849365234375,24.56211235799689]
                                       , [-81.507568359375,24.669482313373848]
                                       , [-81.34002685546875,24.649513490158643]
                                       , [-81.04339599609375,24.731864277701714]
                                       , [-80.771484375,24.856534339310674]
                                       , [-80.53253173828124,24.991036982463747]
                                       , [-80.39794921875,25.147770882723563]
                                       ]
                       }
                     );
```

The first variable is pretty obvious, it is our distance from the line, in meters, we want our data to lie in. In this case we will use 500 meters as our value.

The second variable, geoLine, holds json data that includes an array of longitudes / latitudes for the line. I've already run the query so, thanks to the magic of my video editing software, I can overlay a line on the screen which corresponds roughly to the array of coordinates in our geoLine variable. As you can see, it roughly covers the route of US Highway 1 from Key West

up to Key Largo.

Next we take the storm events, filter out the rows with either an empty long or lat, then reduce to just the three columns we need using project.

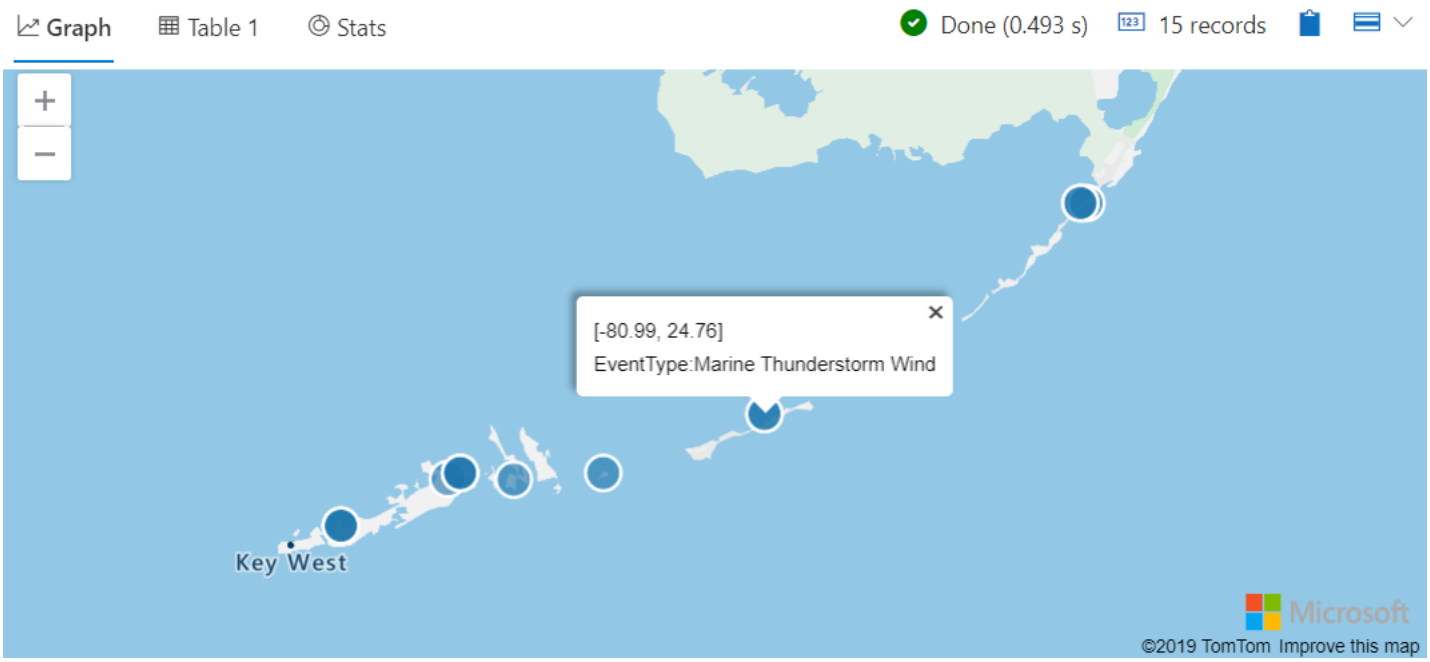
```
StormEvents
  | where isnotempty( BeginLat ) and isnotempty( BeginLon )
  | project BeginLon, BeginLat, EventType
  | where geo_distance_point_to_line( BeginLon, BeginLat, geoLine ) < distanceFromLine
  | render scatterchart with (kind = map)
```

Now we get to the heart of this demo, and use the `geo_distance_point_to_line` function to filter the results down to just those storm events whose distance from the line is less than 500 meters.

Finally we render the results as a scatter chart overlaid onto a map.

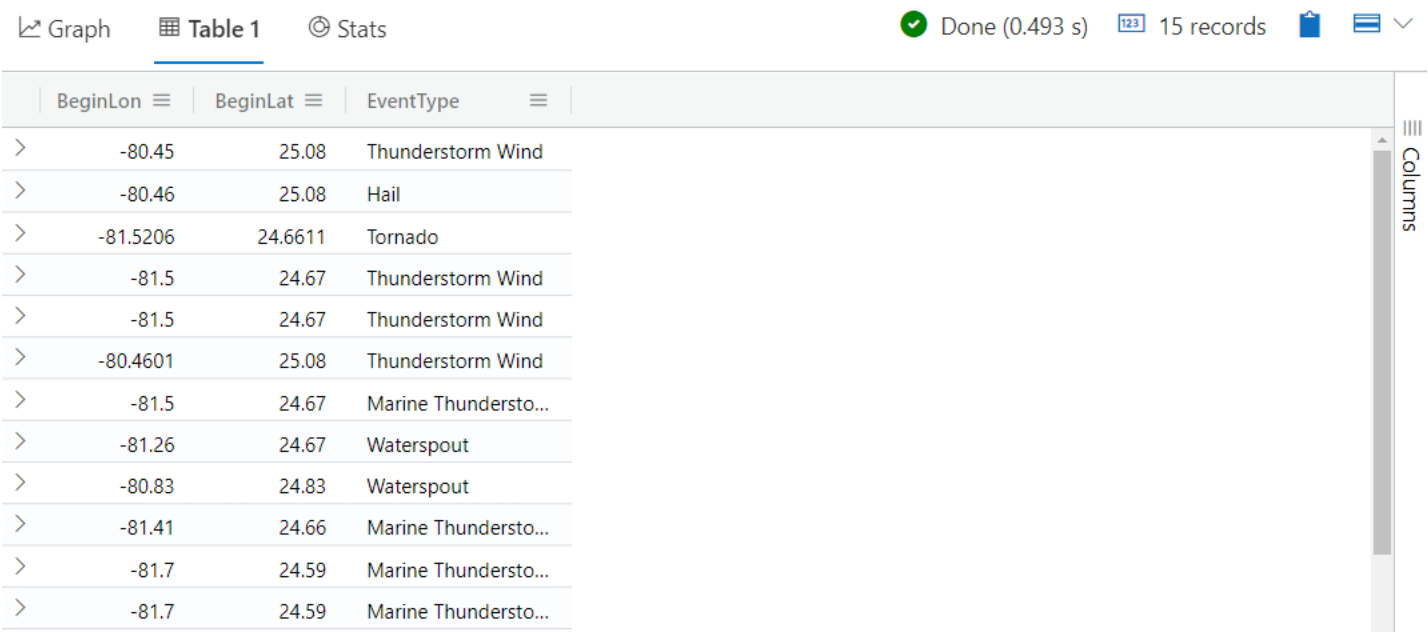
1.3.2 Analyzing the Output

Here are the results, laid on a nice map.



Note the circles of storm events along the geographic line through the Florida keys. You'll see we've clicked on one of the events to display its coordinates along with the type of event.

If you wish to see the text output, then simply click on the Table tab as we did in the previous example with geographic circles.



BeginLon	BeginLat	EventType
-80.45	25.08	Thunderstorm Wind
-80.46	25.08	Hail
-81.5206	24.6611	Tornado
-81.5	24.67	Thunderstorm Wind
-81.5	24.67	Thunderstorm Wind
-80.4601	25.08	Thunderstorm Wind
-81.5	24.67	Marine Thundersto...
-81.26	24.67	Waterspout
-80.83	24.83	Waterspout
-81.41	24.66	Marine Thundersto...
-81.7	24.59	Marine Thundersto...
-81.7	24.59	Marine Thundersto...

Summary

In this demo, we examined functions for returning results that are within a specified geographic distance from either a single point, or along a geographic line.

Module 2 - Geoclustering

Demo 2 - Geofencing

2.1 - Polygon Overview

The focus of this demo is the function `geo_point_in_polygon`. You can think of a polygon like a fence that wraps around a geographic area.

2.1.1 Examining the Code

In this first example, we will define a single polygon, that is a single area we want to plot data for. We'll start the query by defining a json variable that holds the coordinates of our polygon.

```
let polygon = dynamic( { "type":"Polygon"
                       , "coordinates":[ [ [-81.06880187988281,24.75306702526595]
                                         , [-81.12510681152344,24.728122241065808]
                                         , [-81.13609313964844,24.691319554166277]
                                         , [-81.09901428222656,24.671978191593258]
                                         , [-81.03858947753905,24.70005337937338]
                                         , [-80.97129821777344,24.718766657061526]
                                         , [-80.92391967773438,24.74558411549905]
                                         , [-80.86624145507812,24.775513050757333]
                                         , [-80.93215942382812,24.79483832122786]
                                         , [-81.06880187988281,24.75306702526595]
                                         ]
                                       ]
                       }
                     );
```

Next, we'll follow a similar pattern that we did in the previous demo.

```
StormEvents
  | where isnotempty( BeginLat ) and isnotempty( BeginLon )
  | project BeginLon, BeginLat, EventType
  | where geo_point_in_polygon(BeginLon, BeginLat, polygon)
  | render scatterchart with (kind = map)
```

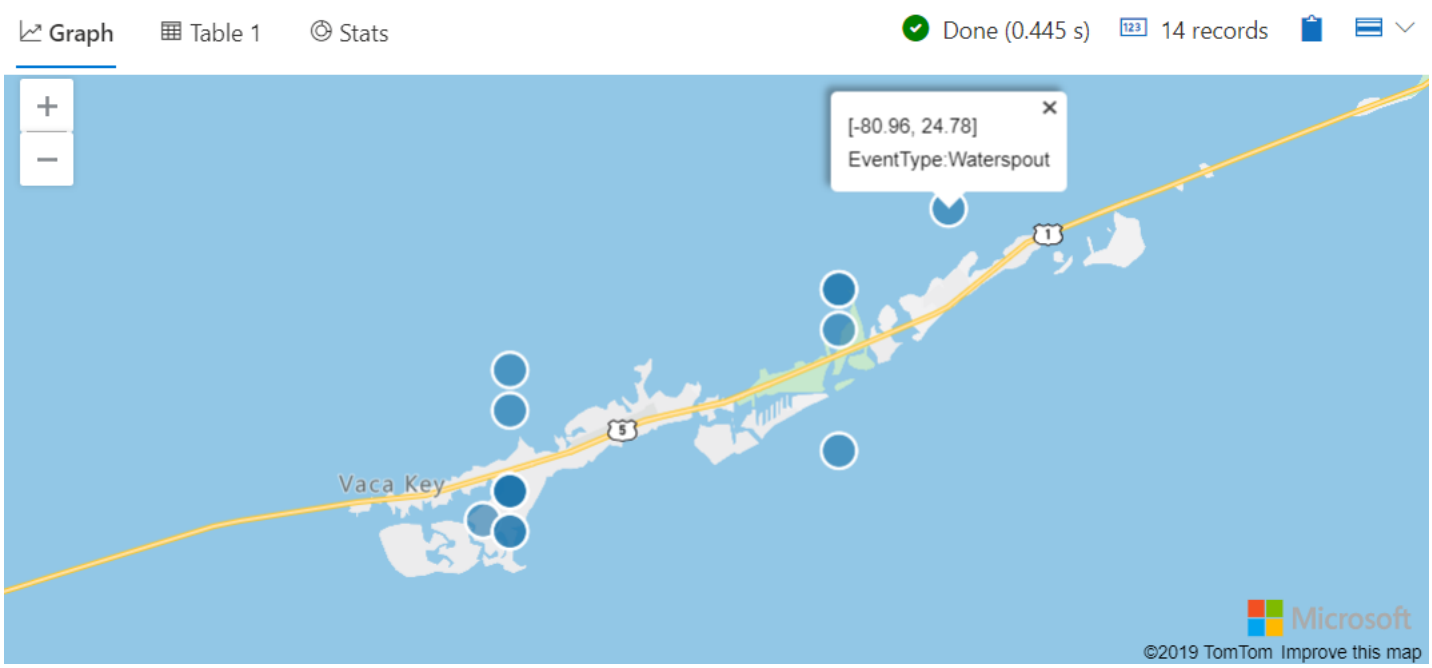
We take our storm events data and filter it through the `where` clause to remove rows that are missing the important lat and long values. Next, we use `project` to reduce the data to just the columns we need.

The next line is where we call the focus for this demo, the `geo_point_in_polygon` function. We pass in the names of the columns from the incoming data that represent the longitude and latitudes. Finally we pass in the polygon to use as our "fence".

In the last step we simply overlay our scatter chart upon a map.

2.1.2 Analyzing the Output

As you can see, Azure Data Explorer renders a nice map with the storm events laid out on it.



Here you can see each event, one of them we've clicked on so it will display the coordinates and the event type.

Clicking the Table tap will produce a text listing of longitudes, latitudes, and event types. As this is similar to what you've seen before, we won't reproduce it here.

2.2 - Multi Polygon Overview

In addition to a single polygon, it's also possible to use `geo_point_in_polygon` with multiple polygons. This is useful when, for example, you want to omit a section from a map for analysis purposes.

2.2.1 Examining the Code

We begin with an extremely long json variable declaration. If you look closely, it actually contains three separate arrays of coordinates within it.

```
let multipolygon = dynamic( { "type":"MultiPolygon"
                            , "coordinates":[ [ [ [-73.991460000000131,40.731738000000206]
                                                , [-73.992854491775518,40.730082566051351]
                                                , [-73.996772,40.725432000000154]
                                                , [-73.997634685522883,40.725786309886963]
                                                , [-74.002855946639244,40.728346630056791]
                                                , [-74.001413,40.731065000000207]
                                                , [-73.996796995070824,40.73736378205173]
                                                , [-73.991724524037934,40.735245208931886]
                                                , [-73.990703782359589,40.734781896080477]
                                                , [-73.991460000000131,40.731738000000206]
                                                ]
                                              ]
                                              ,
                                              [
                                                [ [-73.958357552055688,40.800369095633819]
                                                , [-73.98143901556422,40.768762584141953]
                                                , [-73.981548752788598,40.7685590292784]
                                                , [-73.981565335901905,40.768307084720796]
                                                , [-73.981754418060945,40.768399727738668]
                                                , [-73.982038573548124,40.768387823012056]
                                                , [-73.982268248204349,40.768298621883247]
                                                , [-73.982384797518051,40.768097213086911]
                                                , [-73.982320919746599,40.767894461792181]
                                                , [-73.982155532845766,40.767756204474757]
                                                , [-73.98238873834039,40.767411004834273]
                                                , [-73.993650353659021,40.772145571634361]
                                                , [-73.99415893763998,40.772493009137818]
                                                , [-73.993831082030937,40.772931787850908]
                                                , [-73.993891252437052,40.772955194876722]
                                                , [-73.993962585514595,40.772944653908901]
                                                , [-73.99401262480508,40.772882846631894]
                                                , [-73.994122058082397,40.77292405902601]
                                                , [-73.994136652588594,40.772901870174394]
                                                , [-73.994301342391154,40.772970028663913]
                                                , [-73.994281535134448,40.77299380206933]
                                                , [-73.994376552751078,40.77303955110149]
                                                , [-73.994294029824005,40.773156243992048]
                                                , [-73.995023275860802,40.773481196576356]
                                                , [-73.99508939189289, 40.773388475039134]
                                                , [-73.995013963716758,40.773358035426909]
                                                , [-73.995050284699261,40.773297153189958]
                                                , [-73.996240651898916,40.773789791397689]
                                                , [-73.996195837470992,40.773852356184044]
                                                , [-73.996098807369748,40.773951805299085]
                                                , [-73.996179459973888,40.773986954351571]
                                                , [-73.996095245226442,40.774086186437756]
                                                , [-73.995572265161172,40.773870731394297]
                                                , [-73.994017424135961,40.77321375261053]
                                                , [-73.993935876811335,40.773179512586211]
                                                , [-73.993861942928888,40.773269531698837]
                                                , [-73.993822393527211,40.773381758622882]
                                                , [-73.993767019318497,40.773483981224835]
                                                , [-73.993698463744295,40.773562141052594]
                                                , [-73.993358326468751,40.773926888327956]
                                                , [-73.992622663865575,40.774974056037109]
                                                , [-73.992577842766124,40.774956016359418]
                                                , [-73.992527743951555,40.775002110439829]
                                                , [-73.992469745815342,40.775024159551755]
                                                , [-73.992403837191887,40.775018140390664]
                                                , [-73.99226708903538, 40.775116033858794]
                                                , [-73.99217809026365, 40.775279293897171]
                                                , [-73.992059084937338, 40.775497598192516]
                                                , [-73.992125372394938, 40.775509075053385]
                                                , [-73.992226867797001, 40.775482211026116]
                                                , [-73.992329346608813, 40.775468900958522]
                                                , [-73.992361756801131, 40.775501899766638]
                                                , [-73.992386042960277, 40.775557180424634]
                                                , [-73.992087684712729, 40.775983970821372]
                                                , [-73.990927174149746, 40.777566878763238]
                                                , [-73.99039616003671,40.777585065679204]
                                                , [-73.989461267506471, 40.778875124584417]
                                                , [-73.989175778438053, 40.779287524015778]
                                                , [-73.988868617400072, 40.779692922911607]
                                                , [-73.988871874499793, 40.779713738253008]
                                                , [-73.989219022880576, 40.779697895209402]
                                                , [-73.98927785904425, 40.779723439271038]
                                                , [-73.989409054180143, 40.779737706471963]
                                                , [-73.989498614927044, 40.779725044389757]
                                                , [-73.989596493388234, 40.779698146683387]
                                                , [-73.989679812902509, 40.779677568658038]
                                                , [-73.989752702937935, 40.779671244211556]
                                                , [-73.989842247806507, 40.779680752670664]
                                                , [-73.990040102120489, 40.779707677698219]
                                                , [-73.990137977524839, 40.779699769704784]
                                                , [-73.99033584033225, 40.779661794394983]
                                                , [-73.990430598697046, 40.779664973055503]
                                                , [-73.990622199396725, 40.779676064914298]
                                                , [-73.990745069505479, 40.779671328184051]
                                                , [-73.990872114282197, 40.779646007643876]
                                                , [-73.990961672224358, 40.779639683751753]
                                                , [-73.991057472829539, 40.779652352625774]
                                                , [-73.991157429497036, 40.779669775606465]
                                                , [-73.991242817404469, 40.779671367084504]
                                                , [-73.991255318289745, 40.779650782516491]
                                                , [-73.991294887120119, 40.779630209208889]
                                                , [-73.991321967649895, 40.779631796041372]
                                                , [-73.991359455569423, 40.779585883337383]
                                                , [-73.991551059227476, 40.779574821437407]
```

```
                                , [-73.99141982585985, 40.779755280287233]
                                , [-73.988886144117032, 40.7798788985329999]
                                , [-73.988939656706265, 40.7799561784403931]
                                , [-73.988926103530844, 40.780059292013632]
                                , [-73.988911680264692, 40.780096037146606]
                                , [-73.988919261468567, 40.780226094343945]
                                , [-73.988381050202634, 40.780981074045783]
                                , [-73.988232413846987, 40.781233144215555]
                                , [-73.988210420831663, 40.781225482542055]
                                , [-73.988140000000143, 40.781409000000224]
                                , [-73.988041288067166, 40.781585961353777]
                                , [-73.98810029382463, 40.781602878305286]
                                , [-73.988076449145055, 40.781650935001608]
                                , [-73.988018059972219, 40.781634188810422]
                                , [-73.987960792842145, 40.781770987031535]
                                , [-73.985465811970457, 40.785360700575431]
                                , [-73.986172704965611, 40.786068452258647]
                                , [-73.986455862401996, 40.785919219081421]
                                , [-73.987072345615601, 40.785189638820121]
                                , [-73.98711901394276, 40.785210319004058]
                                , [-73.986497781023601, 40.785951202887254]
                                , [-73.986164628806279, 40.786121882448327]
                                , [-73.986128422486075, 40.786239001331111]
                                , [-73.986071135219746, 40.786240706026611]
                                , [-73.986027274789123, 40.786228964236727]
                                , [-73.986097637849426, 40.78605822569795]
                                , [-73.985429321269592, 40.785413942184597]
                                , [-73.985081137732209, 40.785921935110366]
                                , [-73.985198833254501, 40.785966552197777]
                                , [-73.985170502389906, 40.78601333415817]
                                , [-73.985216218673656, 40.786030501816427]
                                , [-73.98525509797993, 40.785976205511588]
                                , [-73.98524273937646, 40.785972572653328]
                                , [-73.98524962933017, 40.785963139855845]
                                , [-73.985281779186749, 40.785978620950075]
                                , [-73.985240032884533, 40.786035858136792]
                                , [-73.985683885242182, 40.786222123919686]
                                , [-73.985717529004575, 40.786175994668795]
                                , [-73.985765660297687, 40.786196274858618]
                                , [-73.985682871922691, 40.786309786213067]
                                , [-73.985636270930442, 40.786290150649279]
                                , [-73.985670722564691, 40.786242911993817]
                                , [-73.98520511880038, 40.7860476692127851
                                , [-73.985211035607492, 40.786039554883686]
                                , [-73.985162639946992, 40.786020999769754]
                                , [-73.985131636312062, 40.786060297019972]
                                , [-73.985016964065125, 40.78601423719563]
                                , [-73.984655078830457, 40.786534741807841]
                                , [-73.985743787901043, 40.786570082854738]
                                , [-73.98589227228328, 40.786426529019593]
                                , [-73.985942854994988, 40.786452847880334]
                                , [-73.985949561556794, 40.78648711396653]
                                , [-73.985812373526713, 40.786616865357047]
                                , [-73.985135209703174, 40.78658761889551]
                                , [-73.984619428584324, 40.786586016349787]
                                , [-73.981952458164173, 40.790393724337193]
                                , [-73.972823037363767, 40.8034280528167561
                                , [-73.971036786332192, 40.805918478839672]
                                , [-73.966701, 40.8041690000001861
                                , [-73.959647, 40.8011560000001131
                                , [-73.958508540159471, 40.800682279767472]
                                , [-73.95853274080838, 40.800491362464697]
                                , [-73.958357552055688, 40.800369095633819]
                                ]
                            ]
                            ,
                            [ [ [-73.943592454622546, 40.782747908206574]
                                , [-73.943648235390199, 40.782656161333449]
                                , [-73.943870759887162, 40.781273026571704]
                                , [-73.94345932494096, 40.780048275653243]
                                , [-73.943213862652243, 40.779317588660199]
                                , [-73.943004239504688, 40.779639495474292]
                                , [-73.942716005450905, 40.779544169476175]
                                , [-73.942712374762181, 40.779214856940001]
                                , [-73.942535563208608, 40.779090956062532]
                                , [-73.942893408188027, 40.778614093246276]
                                , [-73.942438481745029, 40.777315235766039]
                                , [-73.942244919522594, 40.777104088947254]
                                , [-73.942074188038887, 40.776917846977142]
                                , [-73.942002667222781, 40.776185317382648]
                                , [-73.942620205199006, 40.775180871576474]
                                , [-73.94285645694552, 40.774796600349191]
                                , [-73.94293043781397, 40.774676268036011]
                                , [-73.945870899588215, 40.771692257932997]
                                , [-73.946618690150586, 40.770933392569561
                                , [-73.948664164778933, 40.768857624399587]
                                , [-73.950069793030679, 40.767025088383498]
                                , [-73.954418260786071, 40.762184104951245]
                                , [-73.95650786241211, 40.760285256574043]
                                , [-73.958787773424007, 40.758213471309809]
                                , [-73.973015157270069, 40.764278692864671]
                                , [-73.955760332998182, 40.787906554459667]
                                , [-73.944023, 40.7829600000003011
                                , [-73.943592454622546, 40.782747908206574]
                                ]
                            ]
                        ]
    }
);
```

For this demo, we're going to move north to the city that never sleeps, New York, and specifically Manhattan. We are going to include three different polygons. One will be just to the west of the historic Central Park, one just to the east, and one south around the East Village area.

Our goal is to plot storm events, but we want to omit the Central Park area. This will let us focus on areas with sky scrapers and omit a natural area like the park.

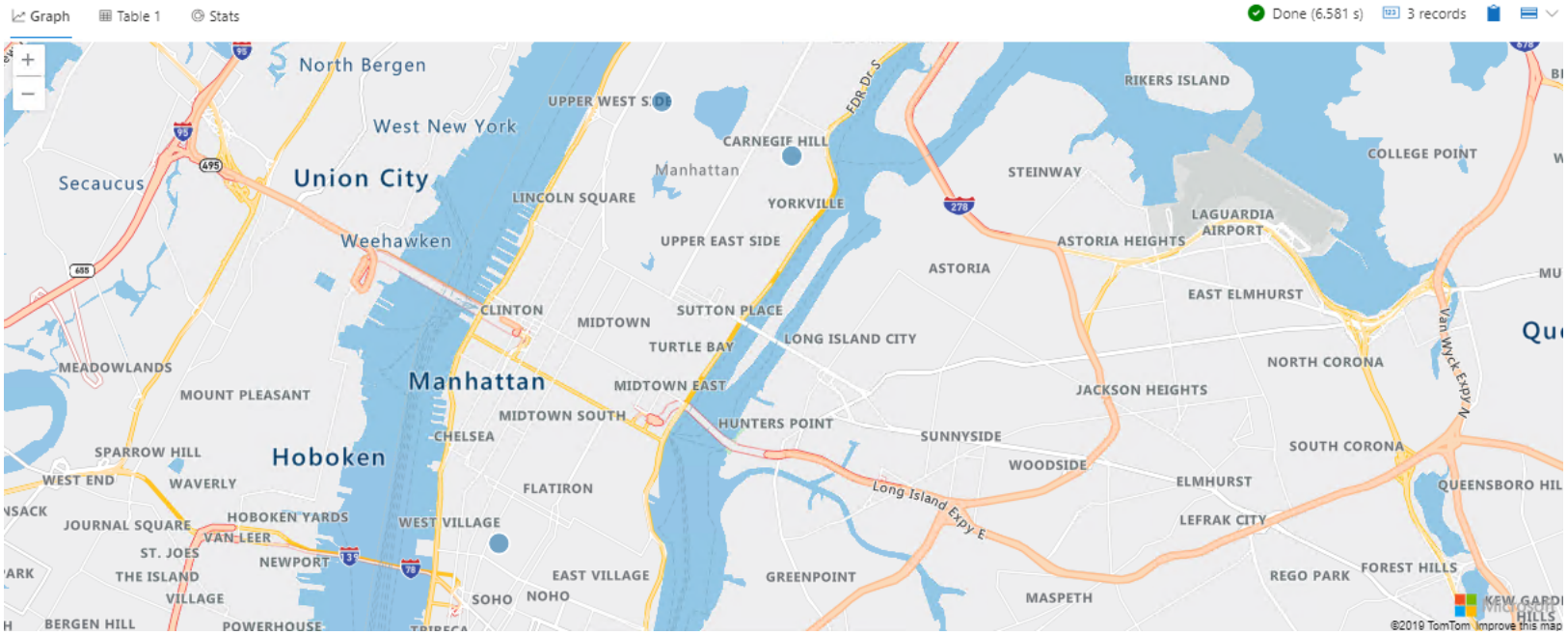
The next section is almost identical to the previous one, except for the name of our polygon variable, `multipolygon`.

```
StormEvents
| where isnotempty( BeginLat ) and isnotempty( BeginLon )
| project BeginLon, BeginLat, EventType
| where geo_point_in_polygon(BeginLon, BeginLat, multipolygon)
| render scatterchart with (kind = map)
```

As a quick recap, we filter out data with missing lats or longs, reduce it to just the data we need, run it through the `geo_point_in_polygon` function, then render as a scatterchart over a map.

2.2.2 Analyzing the Output

Here is the results of our query.



Looking closely, you'll see one storm plotted in the upper middle, beside the Upper West Side. This was the area defined in one of our polygons, just to the west of Central Park.

There's a second storm event just under Carnegie Hill, in the area defined by the polygon placed just to the east of Central Park.

Finally, theres another event that was located in our third polygon, partway between the West Village and Soho.

Summary

When you need to search for events inside a defined area, or more than one area, the `geo_point_in_polygon` function will quickly become your favorite tool.

Module 2 - Geoclustering

Demo 3 - Clustering

3.1 Clustering Based on S2 Cells - Overview

In the first example, we'll see how to perform geoclustering based on S2 cells. Now, you may be wondering "What in the world is an S2 cell?"

That's a great question, let's head over to the S2Geometry website to look at their definition.

In S2Geometry, the earth is divided into cells. Each cell is further subdivided into cells, and each of those into even more cells. S2 Cells can go up to 30 levels deep.

In looking at the map on their website, your reaction may be "But wait, those aren't square cells, their curved!" True, but remember with maps like this we are taking a sphere and flattening it out.

The curves are similar to what you see on a map when you are looking at an airplane flight bringing your mother in law for a visit. On the map it looks like the plane is taking a curved route, but over the planet the plane is flying in a straight line.

So how can we plot our data, in this case storm data, using the S2 Cell system? Well lucky for us, Kusto has a couple of functions just for that. Let's take a look!

3.1.1 Examining the Code

As with our other demos, we'll start by defining a json variable with a set of geographic coordinates, filled with longitudes and latitudes. These coordinates define the state of California.

```
let california = dynamic( { "type":"Polygon"
                          ,   "coordinates":[ [ [-123.233256,42.006186]
                                              , [-122.378853, 42.011663]
                                              , [-121.037003, 41.995232]
                                              , [-120.001861, 41.995232]
                                              , [-119.996384, 40.264519]
                                              , [-120.001861, 38.999346]
                                              , [-118.71478, 38.101128]
                                              , [-117.498899, 37.21934]
                                              , [-116.540435, 36.501861]
                                              , [-115.85034, 35.970598]
                                              , [-114.634459, 35.00118]
                                              , [-114.634459, 34.87521]
                                              , [-114.470151, 34.710902]
                                              , [-114.333228, 34.448009]
                                              , [-114.136058, 34.305608]
                                              , [-114.256551, 34.174162]
                                              , [-114.415382, 34.108438]
                                              , [-114.535874, 33.933176]
                                              , [-114.497536, 33.697668]
                                              , [-114.524921, 33.54979]
                                              , [-114.727567, 33.40739]
                                              , [-114.661844, 33.034958]
                                              , [-114.524921, 33.029481]
                                              , [-114.470151, 32.843265]
                                              , [-114.524921, 32.755634]
                                              , [-114.72209, 32.717295]
                                              , [-116.04751, 32.624187]
                                              , [-117.126467, 32.536556]
                                              , [-117.24696, 32.668003]
                                              , [-117.252437, 32.876127]
                                              , [-117.329114, 33.122589]
                                              , [-117.471515, 33.297851]
                                              , [-117.7837, 33.538836]
                                              , [-118.183517, 33.763391]
                                              , [-118.260194, 33.703145]
                                              , [-118.413548, 33.741483]
                                              , [-118.391641, 33.840068]
                                              , [-118.566903, 34.042715]
                                              , [-118.802411, 33.998899]
                                              , [-119.218659, 34.146777]
                                              , [-119.278905, 34.26727]
                                              , [-119.558229, 34.415147]
                                              , [-119.875891, 34.40967]
                                              , [-120.138784, 34.475393]
                                              , [-120.472878, 34.448009]
                                              , [-120.64814, 34.579455]
                                              , [-120.609801, 34.858779]
                                              , [-120.670048, 34.902595]
                                              , [-120.631709, 35.099764]
                                              , [-120.894602, 35.247642]
                                              , [-120.905556, 35.450289]
                                              , [-121.004141, 35.461243]
                                              , [-121.168449, 35.636505]
                                              , [-121.283465, 35.674843]
                                              , [-121.332757, 35.784382]
                                              , [-121.716143, 36.195153]
                                              , [-121.896882, 36.315645]
                                              , [-121.935221, 36.638785]
                                              , [-121.858544, 36.6114]
                                              , [-121.787344, 36.803093]
                                              , [-121.929744, 36.978355]
                                              , [-122.105006, 36.956447]
                                              , [-122.335038, 37.115279]
                                              , [-122.417192, 37.241248]
                                              , [-122.400761, 37.361741]
                                              , [-122.515777, 37.520572]
                                              , [-122.515777, 37.783465]
                                              , [-122.329561, 37.783465]
                                              , [-122.406238, 38.15042]
                                              , [-122.488392, 38.112082]
                                              , [-122.504823, 37.931343]
```

```
                                    , [-122.701993, 37.893004]
                                    , [-122.937501, 38.029928]
                                    , [-122.97584, 38.265436]
                                    , [-123.129194, 38.451652]
                                    , [-123.331841, 38.566668]
                                    , [-123.44138, 38.698114]
                                    , [-123.737134, 38.95553]
                                    , [-123.687842, 39.032208]
                                    , [-123.824765, 39.366301]
                                    , [-123.764519, 39.552517]
                                    , [-123.85215, 39.831841]
                                    , [-124.109566, 40.105688]
                                    , [-124.361506, 40.259042]
                                    , [-124.410798, 40.439781]
                                    , [-124.158859, 40.877937]
                                    , [-124.109566, 41.025814]
                                    , [-124.158859, 41.14083]
                                    , [-124.065751, 41.442061]
                                    , [-124.147905, 41.715908]
                                    , [-124.257444, 41.781632]
                                    , [-124.213628, 42.000709]
                                    , [-123.233256, 42.006186]
                                    ]
                                  ]
                    }
                );
```

In the next line, we're going to create a new variable, `s2Level`.

```
let s2Level = 7;
```

If you recall from the overview, I mentioned that each S2 cell could be subdivided into more S2 cells. These cells can go up to 30 levels deep. Here we're going to use this variable to indicate we want to go down to the 7th level of cells.

Next, we'll use the same `geo_point_in_polygon` function we saw in the last demo to limit our storm results to just those that occurred in the state of california.

```
StormEvents
  | project BeginLon, BeginLat, EventType
  | where geo_point_in_polygon(BeginLon, BeginLat, california)
```

In the next line of our query, we're going to use `summarize` to get counts of our data.

```
  | summarize eventCount = count() by EventType
            , hash = geo_point_to_s2cell(BeginLon, BeginLat, s2Level)
```

We will group first by the event type (Tornados, Heavy Rain, etc.), assigning the column name of `eventCount` to the count.

The second part of our summary will be the S2 cell. To get the S2 cell, we will employ the `geo_point_to_s2cell` function. In it we pass the column names for our longitude and latitude. The final value is the level we want to drill down to, here indicated by the value in `s2Level`.

The `geo_point_to_s2cell` function returns a hashed string with the S2 cell information, which we will assign to the new column appropriately named `hash`.

In the next line of code, we'll project our new columns we'll need to render the data in a map.

```
  | project geo_s2cell_to_central_point(hash), EventType, eventCount
```

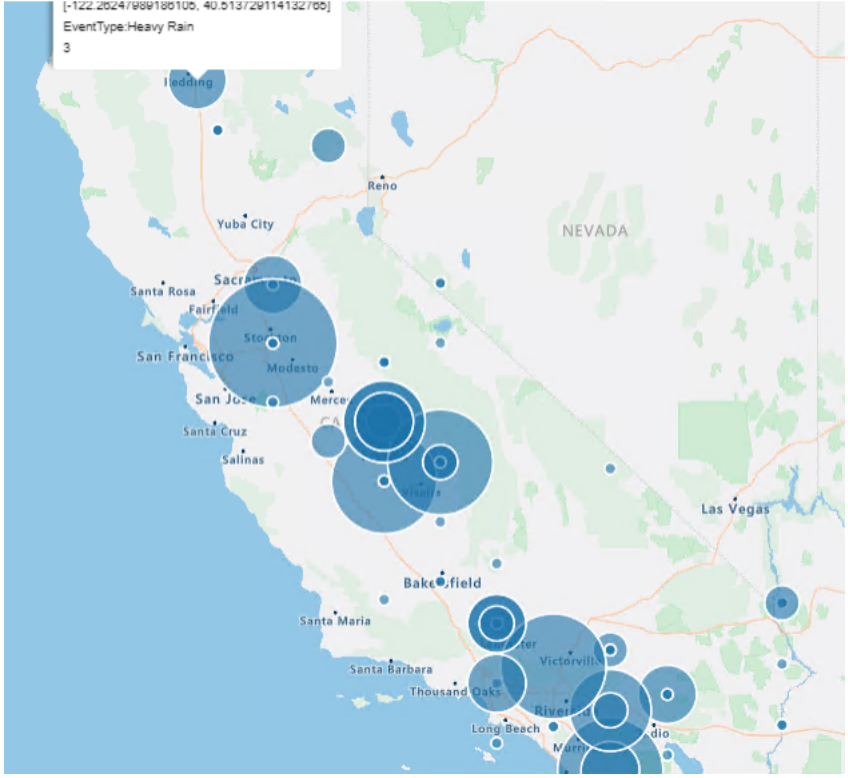
The first column employs the `geo_s2cell_to_central_point` function. This takes the hash value for the S2 cell, and converts it to the long/lat for the central point in the S2 cell. We then add the event type and count to the projected output.

With this data we can now render the data as a series of pie charts, overlaid on the map of California.

```
  | render piechart with (kind=map)
```

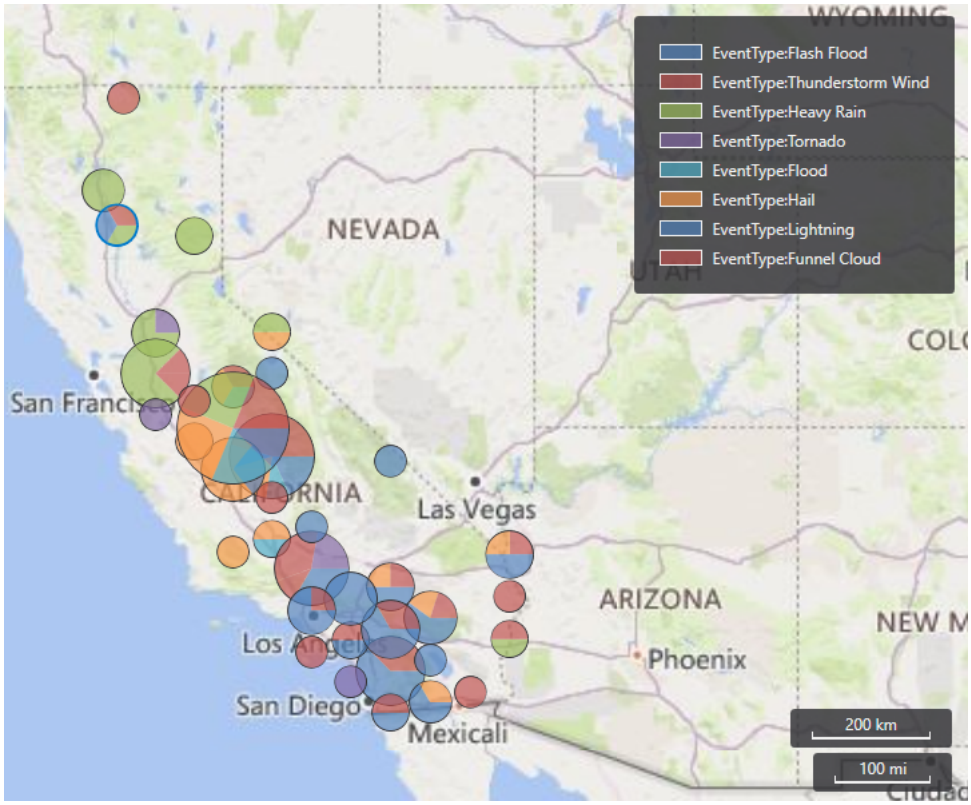
3.1.2 Analyzing the Output

Let's take a look at the resulting map. Or I should say maps, as the maps render differently depending on which tool you use to execute the query. Here is the map as rendered in the Azure Data Explorer webpage.



If the map has a single dot, then it represents a single storm event. If there are smaller dots within a bigger dot, they represent smaller portions of the counts, similar to the way a normal pie chart subdivides its data. You can click on any of the dots to get more information on what type of event it was, as well as the count of that event.

Now let's see what the output looks like using the Kusto one click desktop application.



In this rendering of the map, you see actual pie charts on the map. There is a helpful color map on the upper right which shows the various event types and their corresponding colors. If you were to click on any of them, a pop up would provide exact numbers for each event type for that location.

3.2 Clustering Based on Geohashing - Overview

As an alternative to using S2 cells for clustering, you can also use a concept known as a *geohash*.

3.2.1 Examining the Code

The code begins by defining the polygon for California. As this is exactly the same as the previous demo we'll skip inclusion of it to keep this document brief.

Next, we define a variable called `accuracy`.

```
let accuracy = 3;
```

We'll cover accuracy more in must a moment, meanwhile the next few lines are the same as the previous demo. We take our storm events table and pipe it through `project` to limit the amount of data we work with. We then use the `geo_point_in_polygon` function to further limit the data to just the state of California.

```
StormEvents
  | project BeginLon, BeginLat, EventType
  | where geo_point_in_polygon(BeginLon, BeginLat, california)
```

In the next statement we use `summarize` to count our events, and subdivide those by the location. The location is determined using the `geo_point_to_geohash` function.

```
  | summarize eventCount = count() by EventType
            , hash = geo_point_to_geohash(BeginLon, BeginLat, accuracy)
```

The function will use the long/lat stored in the BeginLon and BeginLat columns to calculate a string hash value for that location. The accuracy value is a number in the range of 1 to 18. The higher the value, the longer and hence more accurate the hash will be.

Conversely, having a larger number may result in many more clusters on the map, many of which are so low as not to be useful in analysis.

Note this value is an optional parameter, if omitted the default value of 5 will be used.

Next, we'll use `project` to output a new set of columns.

```
  | project geo_geohash_to_central_point(hash), EventType, eventCount
```

The last two are pretty obvious. it is the event type and count of those events. With the `geo_geohash_to_central_point`, we take our hash value and convert it to a set of long/lat coordinates.

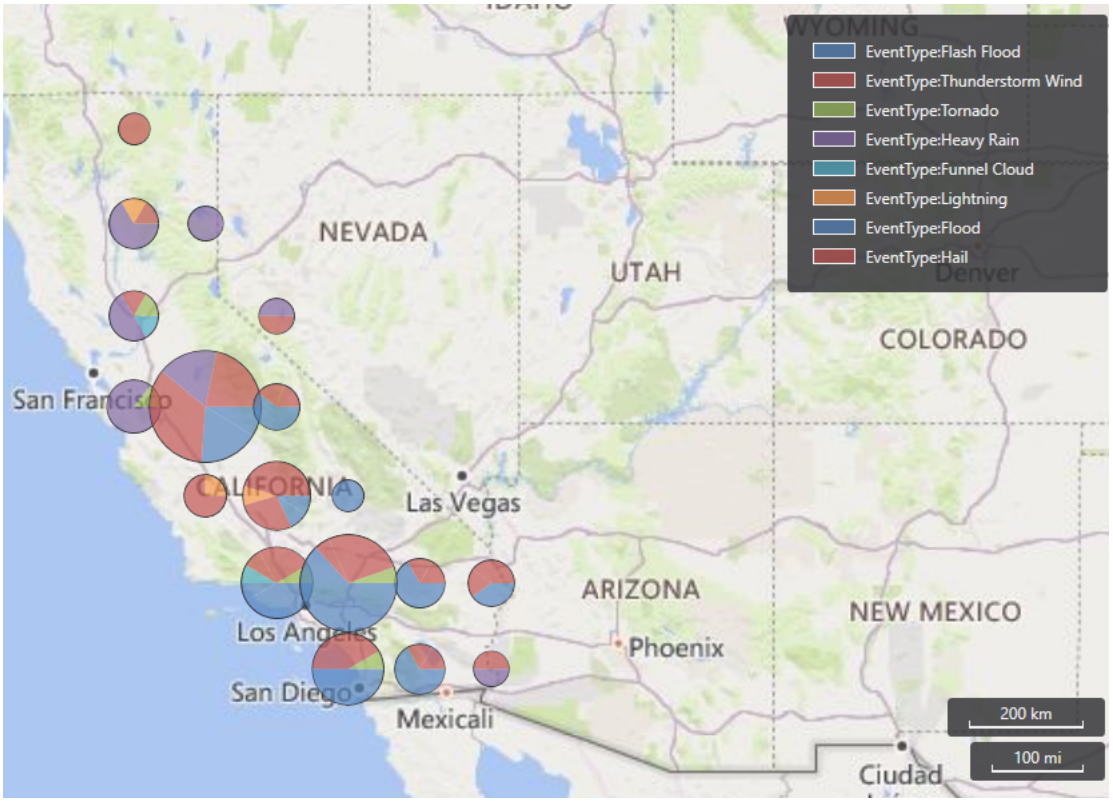
With that done, we can render a nice map overlaid with pie charts.

```
  | render piechart with (kind=map)
```

3.2.2 Analyzing the Output

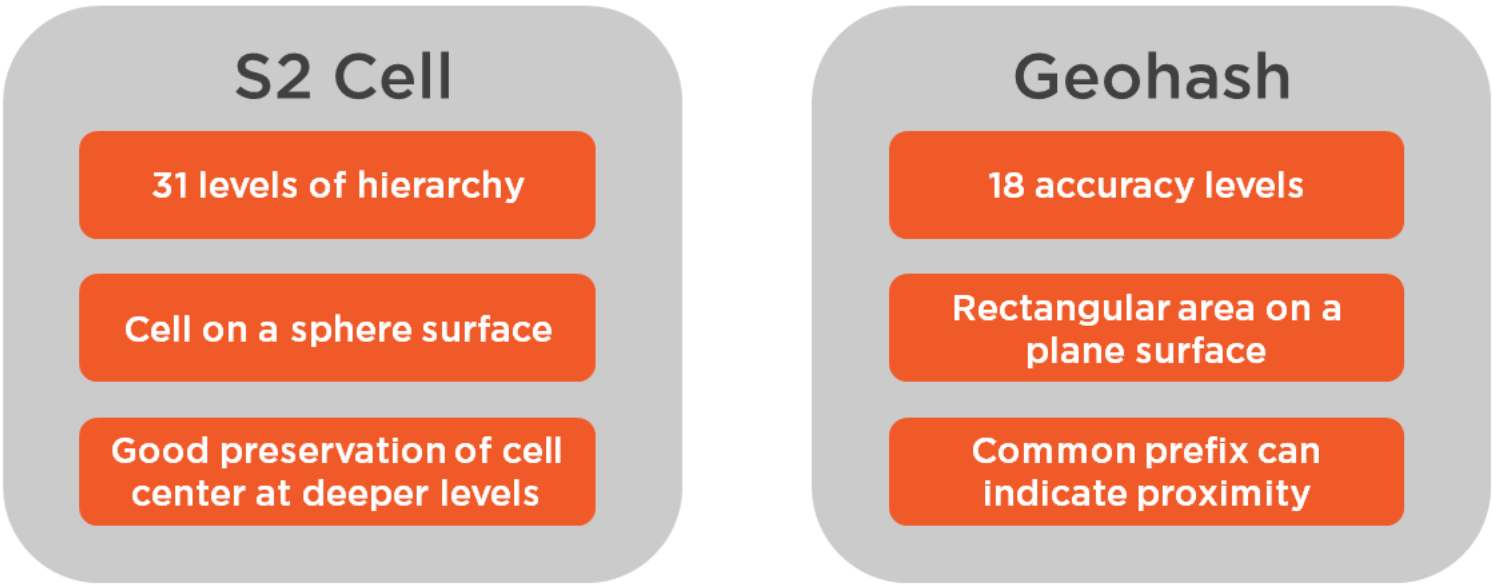
The same differences with map rendering in the ADX website versus the Kusto desktop app also exist in this output, concentric blue dots on the ADX site and actual pie charts in the desktop app.

For brevity, we'll just show the output of the Kusto app here.



S2 Cells vs Geohash

At this point you are likely wondering what is the difference between using the S2 Cell method versus Geohashing for geographic analysis. This image provides a good overview.



The first difference comes in the levels of accuracy. S2 cells can go all the way to 31 levels of accuracy, as contrasted to the 18 levels in a geohash.

Second is the method by which their cells are plotted on a map. As you saw in the beginning of the course, S2 uses a spherical map, so cells appear to have a slightly curved shape when plotted on a flat map. Contrast this with geohashing, which uses rectangular cells laid out on a plane, rather than a sphere.

S2 Cells do better job of preserving the center point of a cell, as compared to a geohash. However, looking at the value of an S2 cell provides no useful information when compared to another S2 value. Geohashes on the other hand, use a system where the first part of the code, or the prefix, can indicate proximity to another code.

In a simplified example, and note these aren't real geohash values, let's say you had three hashes of aaa1, aaa2, and abc1. The two hashes with aaa in the beginning would be closer together geographically than the hash that begins with abc.

Summary

Using the techniques in this section of the module, you can provide two different methods for clustering your data within a geographic fence. Here, we fenced off the state of California, and created clusters based on the event types and number of those events.

Module 2 - Geoclustering

Demo 4 - Geospatial Joins

Overview

Geospatial joins are a technique that allow you to join data from two different data sets based on geographic data. In this demo, we'll see how to join a list of US States to the Storm Events data, so we can aggregate counts of events for each state.

Examining the Code

We begin the query by assigning a variable with a level value. If you recall from the last demo, S2 Cell functions allow you to specify the level we want to drill down to.

```
let s2Level = 5;
```

We now start with the first of our two data sets, the `US_States` table. The data we want is in the `features` column. It contains data in json format, we'll be using `project` to create new columns based on the data in the json.

```
US_States
  | project State = features.properties.NAME
          , polygon = features.geometry
```

The name of the state is easy to understand, it is one of the properties in the features column. The second thing we are interested in is the geometry. Let's again refer back to the previous demo where we defined the geography for the state of California. It was a long list of lat/long coordinates. The geography for each state is stored in the `US_States` table in this geometry property.

Here we extract both properties, the name and geometry, and use `project` to move them into new columns.

Now we need to add a new column, which we'll accomplish using `extend`.

```
  | extend covering = geo_polygon_to_s2cells(polygon, s2Level)
  | mv-expand covering to typeof(string)
```

We'll use the `geo_polygon_to_s2cells` to create an array of S2 cell values, based on the polygon that defines a state and the level we want to drill down to. This array of S2 cells will be stored in the covering column name. After creating the column we'll use `mv-expand` to convert it to a string.

Now we can setup our join.

```
  | join kind = inner hint.strategy = broadcast
  (
    StormEvents
    | project BeginLon, BeginLat
    | extend covering = geo_point_to_s2cell(BeginLon, BeginLat, s2Level)
  ) on covering
```

We start by declaring the join type, an inner join. Using the hint will materialize the left side then broadcast it to all cluster nodes. Remember, Azure Data Explorer is processing these rows in parallel, to do so it spreads the workload across multiple nodes. By using the hint, we make sure all nodes have a full copy of the left side of the data.

For the query we are joining to, we take the storm events and limit it to just the longitude and latitude columns.

Next, we add a new column by using the `geo_point_to_s2cell` function. It will calculate the value of the S2 cell for this particular long/lat combination.

The join is made when the S2 value for this storm event is found in the S2 array for the entire state.

Next we need to apply a filter.

```
  | where geo_point_in_polygon(BeginLon, BeginLat, polygon)
```

Sometimes you'll have an occurrence where the S2 value isn't totally accurate for our needs. Here's a simple example, let's say you have a storm event in the panhandle area of Florida, very close to the Alabama state line. While the event was in Florida, the center of the S2 cell may be across the line in Alabama. This where clause will filter out these anomalies.

Finally, we will summarize our data into counts by state, then add an order by so we get the list in alphabetical order.

```
  | summarize NumberOfEvents = count() by tostring(State)
  | order by State asc
```

Analyzing the Output

Looking at the output, we'll see a list of all US states, along with the number of storm events in that state.

For brevity we only show the first 5 below.

State	NumberOfEvents
Alabama	691
Alaska	17
Arizona	258
Arkansas	709
California	131

Summary

As you can see, performing joins based on geographic locations turns out to be fairly easy, using the functions to convert your long/lat coordinates into S2 cells or points in an S2 cell.

With minor variations you can find other uses for geographic joins. In the downloadable samples we've included samples of using this technique with charts, as well as demonstrating its use at different levels such as counties instead of states.

Module 3 - Performing Diagnostic and Root Cause Analysis

Demo 1 - Clustering a Single Recordset

Overview

In this demo, we're going to see how to cluster a single record set of exceptions. We'll create a time series chart to quickly visualize spike anomalies and drill down to one of them to further investigate.

A Quick Overview of Our Data

First, let's get a count of rows for our sample table to see the volume we're dealing with.

```
demo_clustering1
  | count
```

As of the time of this writing, there are 1,023,652 rows in the demo_clustering1 table. (Please note this value may vary as Microsoft makes updates to the sample data.)

If we want to see a sample of the data, we can run the following query.

```
demo_clustering1
  | take 5
```

This returns the following data.

PreciseTimeStamp	Region	ScaleUnit	DeploymentId	Tracepoint	ServiceHost
2016-08-25 13:37:30.1013530	ncus	su1	8d0625ea0182482da41c422a4021813c	520010	610137c1-1fe1-417b-827d-de6b43b8c689
2016-08-25 13:37:30.1559327	ncus	su1	8d0625ea0182482da41c422a4021813c	512005	00000000-0000-0000-0000-000000000000
2016-08-25 13:38:17.0655320	ncus	su1	8d0625ea0182482da41c422a4021813c	512005	00000000-0000-0000-0000-000000000000
2016-08-25 13:38:17.1807215	ncus	su1	8d0625ea0182482da41c422a4021813c	520010	610137c1-1fe1-417b-827d-de6b43b8c689
2016-08-25 13:38:17.5728823	ncus	su1	8d0625ea0182482da41c422a4021813c	512005	00000000-0000-0000-0000-000000000000

The precise meaning for each column is not important for this demo, simply understand each row represents a service exception in our (mythical) Azure environment. The column of importance to us is the `PreciseTimeStamp`, which holds the exact time the exception occurred.

Charting Exceptions Over Time

In this section, we're going to render these exceptions as a timechart. Doing so allows us to visually look for spikes, in other words points in time where we had a high number (or peaks) of service exceptions.

The code for this is relatively straight forward.

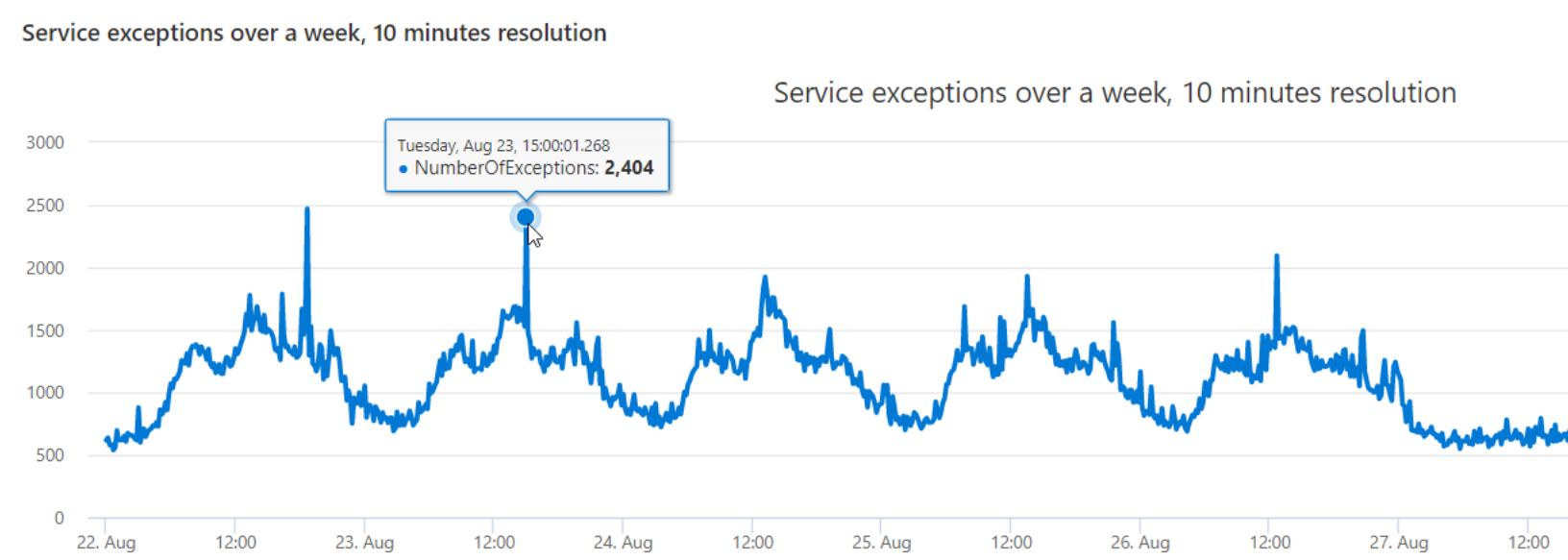
```
let min_t = toscalar(demo_clustering1 | summarize min(PreciseTimeStamp));
let max_t = toscalar(demo_clustering1 | summarize max(PreciseTimeStamp));
demo_clustering1
  | make-series NumberOfExceptions=count()
              on PreciseTimeStamp
            from min_t to max_t step 10m
  | render timechart
      with ( title="Service exceptions over a week, 10 minutes resolution" )
```

In the first two lines, we use summarize to calculate the min and max timestamps from our sample data, and save them into the `min_t` and `max_t` variables.

Next, we pipe our `demo_clustering1` table into the `make_series` function. If you want a more in-depth discussion on make-series I'll refer you to the next module, Time Series Analysis 1.

Here, we are creating a count aggregation and storing it in a new column `NumberOfExceptions`. Our aggregation is done over the `PreciseTimeStamp` column, its boundaries are from `min_t` to `max_t` times variables, and the `step 10m` clause defines the size of the aggregation blocks to 10 minutes. If we were to start on August 23rd at midnight, the first block, or bin, would count everything from 12:00 (included) to 12:10 (excluded). The next block would be from 12:10 to 12:20, and so on.

Finally we render the results as a timechart, providing a nice title for it. Here's the result.



In reviewing the data, we see two spikes towards the beginning of our chart. For our example, let's pretend we have already looked at the first one, and are now focused on the second spike.

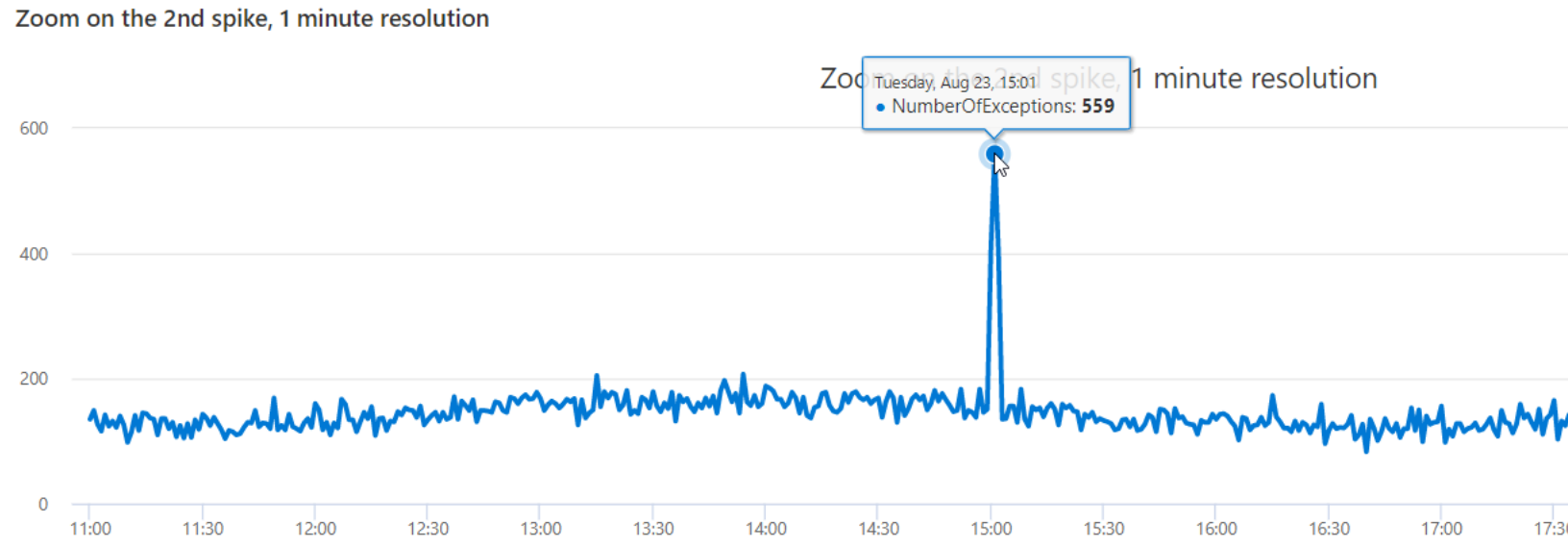
If we hover the mouse over the top of the spike, we'll see the details of it, including the fact it occurred on 23 August at 15:00. This gives us a starting point to dig deeper, which we'll do in the next example.

Digging Deeper into the Exceptions Spike

This query will be similar to the first.

```
let min_t=datetime(2016-08-23 11:00);
demo_clustering1
  | make-series NumberOfExceptions=count()
               on PreciseTimeStamp
            from min_t to min_t+8h step 1m
  | render timechart
      with ( title="Zoom on the 2nd spike, 1 minute resolution" )
```

Here we want to check whether it's a narrow or wide spike, so we "zoom in" by creating a new time series around the spike, from 4 hours before it to 4 hours after it, in 1 minute step size (10x the previous 10m resolution). We can now see that it's indeed a sharp and narrow (2 minutes) spike



We now have the resolution to see the exact minute our spike ocurred, 15:01. Let's construct a query to return the exceptions that occured during this 2 minutes spike for further analysis.

Looking at the Detail Data

Now that we know the spike occurred at 15:01, we'll construct a query to grab data for one minute to either side, 15:00 to 15:02.

```
let min_peak_t=datetime(2016-08-23 15:00);
let max_peak_t=datetime(2016-08-23 15:02);
demo_clustering1
  | where PreciseTimeStamp between(min_peak_t..max_peak_t)
```

This query returned 972 rows x 6 columns (based on the sample data at the time of this writing). Let's see few rows here. Note that commonly you will have much bigger result sets, containing thousands or even millions of rows and plenty of columns.

PreciseTimeStamp	Region	ScaleUnit	DeploymentId	Tracepoint	ServiceHost

PreciseTimeStamp	Region	ScaleUnit	DeploymentId	Tracepoint	ServiceHost
2016-08-23 15:00:00.5050216	scus	su5	9dbd1b161d5b4779a73cf19a7836ebd6	10007006	a7292804-72d3-4ad1-aac5-3c52c9176a00
2016-08-23 15:00:01.4303398	scus	su3	90d3d2fc7ecc430c9621ece335651a01	12040111	00000000-0000-0000-0000-000000000000
2016-08-23 15:00:01.4318226	scus	su3	90d3d2fc7ecc430c9621ece335651a01	12040111	00000000-0000-0000-0000-000000000000
2016-08-23 15:00:01.4955374	weu	su4	be1d6d7ac9574cbc9a22cb8ee20f16fc	12040111	00000000-0000-0000-0000-000000000000
2016-08-23 15:00:01.5013331	weu	su4	be1d6d7ac9574cbc9a22cb8ee20f16fc	12040111	00000000-0000-0000-0000-000000000000

As this is a small result set you can review it, try to look on common values, correlations or other reasons that might explain the spike in exceptions.

Summary

In this demo, we saw a dataset of service exceptions over 1 week. Using basic time series analysis we were able to detect anomalous spikes. We drilled down to a specific spike and review the exception records during that spike, attempting to guess its root cause. In the following parts we are going to see clustering plugins that will help us to find the root cause automatically.

Module 3 - Performing Diagnostic and Root Cause Analysis

Demo 2 - Using Autocluster

Overview

The autocluster plugin is deceptively simple and incredibly powerful. It will analyze a large dataset, and return a short list of patterns, where each pattern is a combination of attributes sharing the same values that were found in this dataset. Let's see how effective it is by analyzing the exceptions' spike from the previous demo, as it is easier to explain when we see the output.

Examining the Code

As mentioned, the query is very simple.

```
let min_peak_t=datetime(2016-08-23 15:00);
let max_peak_t=datetime(2016-08-23 15:02);
demo_clustering1
  | where PreciseTimeStamp between(min_peak_t..max_peak_t)
  | evaluate autocluster()
```

We start as we did in the last demo, only now we pipe it to `evaluate autocluster()`, which will automatically extract interesting patterns for us.

Analyzing the Output

Here is the output of the query.

SegmentId	Count	Percent	PreciseTimeStamp	Region	ScaleUnit	DeploymentId	Tracepoint	ServiceHost
0	639	65.7407407407407		eau	su7	b5d1d4df547d4a04ac15885617edba57		e7f60c5d-4944-42b3-922a-92e98a8e7dec
1	94	9.67078189300411		scus	su5	9dbd1b161d5b4779a73cf19a7836ebd6		
2	82	8.43621399176955		ncus	su1	e24ef436e02b4823ac5d5b1465a9401e		
3	68	6.99588477366255		scus	su3	90d3d2fc7ecc430c9621ece335651a01		
4	55	5.65843621399177		weu	su4	be1d6d7ac9574cbc9a22cb8ee20f16fc		

Under the count column we see there are 639 rows that are contained in the pattern of eau Region, su7 ScaleUnit, DeploymentId b5d1d4df547d4a04ac15885617edba57, and ServiceHost of e7f60c5d-4944-42b3-922a-92e98a8e7dec. This pattern represents 65.74% of the rows in our data. Likewise, subsequent rows represent additional patterns.

Be aware that some rows might be represented in few (partially overlapping) patterns , and some rows are not represented in any pattern. The goal of autocluster is to transform a big table to a very small list of informative and divergent patterns, thus presenting the user significant multi-dimensional patterns for quick and efficient drill down and further investigation for the root cause.

If you see data that looks interesting, you could refine the query to focus in on those attributes. For example you may want to add a `where ServiceHost = 'e7f60c5d-4944-42b3-922a-92e98a8e7dec'` clause prior to the autocluster to look for a new clustering pattern within that one service host.

Summary

As you can see, while simple, autocluster allows you to easily find non-trivial patterns, which are combinations of commonly occurring attributes, within your data.

Module 3 - Performing Diagnostic and Root Cause Analysis

Demo 3 - Using Basket

Overview

Basket analysis is similar to autocluster, in that it looks for patterns in the data. The difference is in the pattern mining algorithm as well as in the amount of data returned. Autocluster is based on a proprietary clustering algorithm, and typically returns a very small set of output patterns. It looks for distinct groups of patterns, returning the most occurring combination in each group.

Basket, on the other hand, is based on the known Apriori algorithm. It returns many more patterns, as it extracts all patterns that contain at least, by default 5% although this is user settable, of the records set.

Examining the Code

```
let min_peak_t=datetime(2016-08-23 15:00);
let max_peak_t=datetime(2016-08-23 15:02);
demo_clustering1
  | where PreciseTimeStamp between(min_peak_t..max_peak_t)
  | evaluate basket()
```

As you can see, the code is almost identical to the autocluster example, the difference being we call basket instead.

Analyzing the Output

SegmentId	Count	Percent	PreciseTimeStamp	Region	ScaleUnit	DeploymentId	Tracepoint	ServiceHost
0	639	65.7407407407407		eau	su7	b5d1d4df547d4a04ac15885617edba57		e7f60c5d-4944-42b3-922a-92e98a8e7dec
1	642	66.0493827160494		eau	su7	b5d1d4df547d4a04ac15885617edba57		
2	324	33.3333333333333		eau	su7	b5d1d4df547d4a04ac15885617edba57	0	e7f60c5d-4944-42b3-922a-92e98a8e7dec
3	315	32.4074074074074		eau	su7	b5d1d4df547d4a04ac15885617edba57	16108	e7f60c5d-4944-42b3-922a-92e98a8e7dec
4	328	33.7448559670782					0	
5	94	9.67078189300411		scus	su5	9dbd1b161d5b4779a73cf19a7836ebd6		
6	82	8.43621399176955		ncus	su1	e24ef436e02b4823ac5d5b1465a9401e		
7	68	6.99588477366255		scus	su3	90d3d2fc7ecc430c9621ece335651a01		
8	167	17.1810699588477		scus				
9	55	5.65843621399177		weu	su4	be1d6d7ac9574cbc9a22cb8ee20f16fc		
10	92	9.46502057613169					10007007	
11	90	9.25925925925926					10007006	
12	57	5.8641975308642						00000000-0000-0000-0000-000000000000

Our output is similar to what we got with autocluster, although here basket found many more patterns, including very similar ones. Interestingly, though using different mining algorithms, both basket and autocluster identified the same top pattern, containing 639 records, 65.74% of the data.

Summary

Basket provides an alternative to autocluster. Autocluster was designed for ad-hoc interactive analysis so it intentionally returns a smaller set of divergent patterns, to avoid "flooding" the user with too many patterns to further investigate.

Basket is the classical implementation of the Apriori algorithm, returning plenty of patterns and better suited for automatic analysis, where all patterns can be further processed by some heuristic or business logic.

Module 3 - Performing Diagnostic and Root Cause Analysis

Demo 4 - Clustering the Difference

Overview

The diffpatterns plugin can characterize differences between two records sets. Frequently it is used to compare records in different time windows, but the split to two records sets can be by another metric within the same time window.

For this demo we'll pick up where the first demo in this module left off. In it, we identified a spike in exceptions that occurred at 15:01 on Aug 23rd. What we want to do now is compare the exceptions' spike that occurred in the two minutes around that incident (15:00 to 15:02) to the exceptions level eight minutes prior to the spike, during which everything seemed to be running OK.

Examining the Code

Let's look at the code needed to execute the diffpatterns plugin to perform this analysis of our spike.

We'll start by defining start and end variables for the anomaly time range.

```
let anomaly_start  = datetime(2016-08-23 15:00);
let anomaly_end    = datetime(2016-08-23 15:02);
```

This date range represents the time during which our anomaly, the spike in exceptions, occurred.

Next we'll define our baseline time range.

```
let baseline_start = datetime(2016-08-23 14:50);
let baseline_end   = datetime(2016-08-23 14:58);
```

This is the start and end times for our baseline, the eight minute window when everything was running fine. Note that we left the 2 minutes before the spike out of the baseline. This is a "gray zone", in most cases leaving it out of the analysis can generate better and cleaner patterns.

```
let split_time=(baseline_end + anomaly_end) / 2.0;
```

Here, we've calculated what is called split time. This is time that marks the end of the baseline and the beginning of our anomalous time. We set it to the middle time point between the end of the baseline and the start of the anomaly, which is 14:59.

Now we can start building the data set.

```
demo_clustering1
  | where (PreciseTimeStamp between(baseline_start..baseline_end))
      or (PreciseTimeStamp between(anomaly_start..anomaly_end))
```

These few lines are straight forward. We take our dataset, and reduce it to contain only exceptions that were within either the baseline or the anomaly time ranges.

Next, we need to create a column which can be used for the diffpatterns plugin to determine if a specific exception came from the baseline time frame, or the anomaly.

```
  | extend AB=iff(PreciseTimeStamp > split_time, 'Anomaly', 'Baseline')
```

The new column will be named `AB` (short for Anomaly/Baseline). We are using the `iff` function, basically an if - then - else statement. If the time on the current row is greater than our `split_time`, we know it is in the anomaly time range, and then the text value **Anomaly** will be set in the `SplitValue` column. Otherwise, the value **Baseline** will be set.

We could have, in fact used any text value for these. A simple A and B, Error and Normal, or Bert and Ernie for example. It's usually best though to use a word that identifies the split, then use that naming standard across all your queries that employ diffpatterns.

Speaking of diffpatterns, we're finally ready to call it.

```
  | evaluate diffpatterns(AB, 'Anomaly', 'Baseline')
```

We start with evaluate as this is a plugin, then call the diffpatterns. The first parameter is the name of the column from the incoming dataset that holds the value to be used for the split. Here we are using the `AB` column we just created.

The next parameters `'Anomaly'` and `'Baseline'` define the 2 subsets to compare. Note that in KQL the input to any operator/plugin is a single table (before the "|"). So the first step of diffpatterns is to take the single input table and split it to 2 subsets: "tbl | where AB == 'Anomaly' " and "tbl | where AB == 'Baseline' ", then it can start mining for patterns and comparing them on both subsets.

In the output, you will see column names like CountA, CountB, PercentA, PercentB, and so forth. The data that comes into the A columns will come from the value in the second parameter. In this case, data with the `AB` value of **Anomaly** will be aggregated into CountA, PercentA, and so on. Where rows have a `AB` value of **Baseline**, the data will be aggregated into the B columns, CountB, PercentB, and so forth.

Analyzing the Output

Running the query, diffpatterns returns few patterns, ordered top down by their significance:

SegmentId	CountA	CountB	PercentA	PercentB	PercentDiffAB	PreciseTimeStamp	Region	ScaleUnit	DeploymentId	Tracepoint	ServiceHost
0	639	21	65.74	1.7	64.04		eau	su7	b5d1d4df547d4a04ac15885617edba57		e7f60c5d-4944-42b3-922a-92e98a8e7dec
1	167	544	17.18	44.16	26.97		scus				

SegmentId	CountA	CountB	PercentA	PercentB	PercentDiffAB	PreciseTimeStamp	Region	ScaleUnit	DeploymentId	Tracepoint	ServiceHost
2	92	356	9.47	28.9	19.43					10007007	
3	90	336	9.26	27.27	18.01					10007006	
4	82	318	8.44	25.81	17.38		ncus	su1	e24ef436e02b4823ac5d5b1465a9401e		
5	55	252	5.66	20.45	14.8		weu	su4	be1d6d7ac9574cbc9a22cb8ee20f16fc		
6	57	204	5.86	16.56	10.69						00000000-0000-0000-0000-000000000000

Let's look at our top pattern:

Attribute	Value
Region	eau
Scale Unit	su7
DeploymentId	b5d1d4df547d4a04ac15885617edba57
Tracepoint	NULL
ServiceHost	e7f60c5d-4944-42b3-922a-92e98a8e7dec

Within our anomaly time period, this pattern contains 639 exceptions, or 65.74% of exceptions during this window. Note that we already found this same pattern with autocluster and with basket. But with diffpatterns we also see how common this pattern was in the baseline period.

Contrasted with the anomalous window, in the baseline window this pattern contains only 21 exceptions, or 1.7% of all exceptions in the baseline. The difference (shown in the PercentDiffAB column) is a whopping 64.04%! Clearly this pattern is the cause of our issues.

Remember the first demo of this module, in which we generated a timechart where we plotted the number of exceptions over time. This allowed us to easily identify a spike in exceptions for a specific time. It was this spike that lead us to use autocluster, basket and now diffpatterns for further investigation.

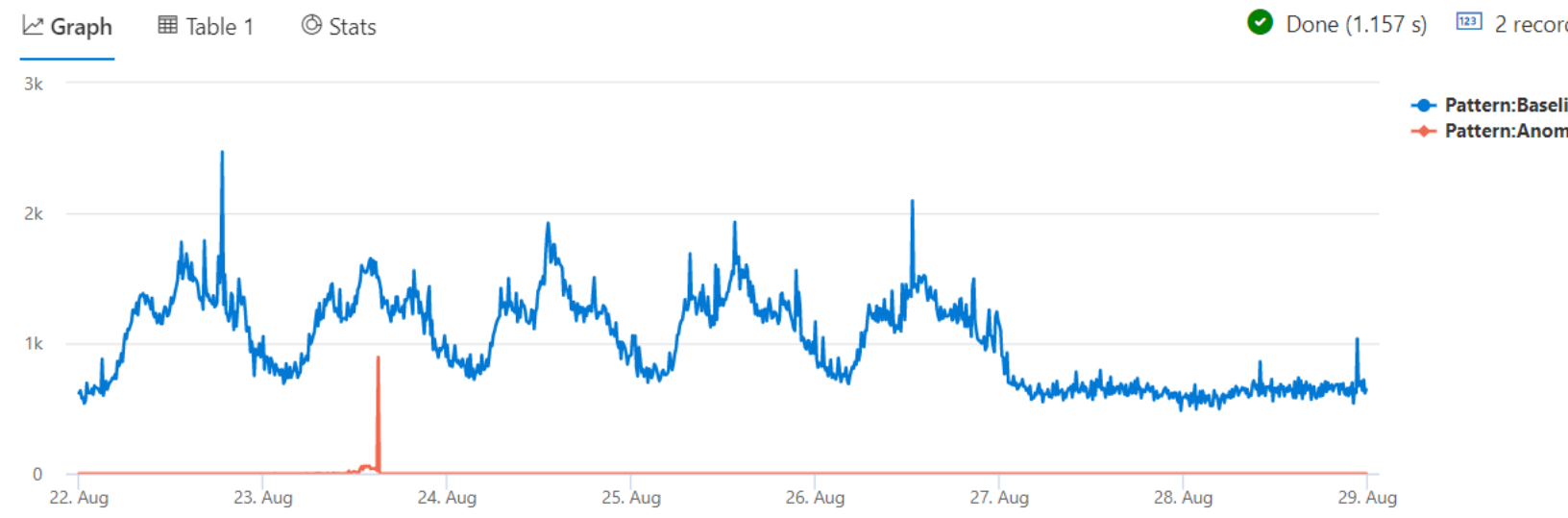
As a final verification step, we can now use the identified pattern and split the data into two lines on our timechart – one for the exceptions count originated from this pattern and another timechart for all other exceptions:

```
let min_t = toscalar(demo_clustering1 | summarize min(PreciseTimeStamp));
let max_t = toscalar(demo_clustering1 | summarize max(PreciseTimeStamp));
demo_clustering1
  | extend Pattern = iff( Region == "eau"
                            and ScaleUnit == "su7"
                            and DeploymentId == "b5d1d4df547d4a04ac15885617edba57"
                            and ServiceHost == "e7f60c5d-4944-42b3-922a-92e98a8e7dec"
                        , "Anomaly"
                        , "Baseline"
                        )
  | make-series Occurrences=count()
           on PreciseTimeStamp
         from min_t to max_t step 10m
           by Pattern
  | render timechart
```

We start by creating variables to hold the max and min time values for the entire dataset.

Next a new column is added to the `demo_clustering1` dataset, `Pattern`. This uses the same `iff` technique from earlier. If the row has the specific attributes then we return a text value of **Anomaly** to the Pattern column. Otherwise, we return the value **Baseline**. Note that the specific attributes we split by are those we got from diffpatterns.

`make-series` is then used to generate 2 time series, one containing the counts of the exceptions that were originated from the Pattern, and one containing counts of all other exceptions, in 10 minutes steps. Piping the output into our `render` yields the following chart:



With the anomaly pattern now isolated into its own line, we can see that it was indeed this pattern that caused the spike in our exceptions.

Summary

Diffpatterns is an advanced machine learning plugin that can mine your data and generate multi-dimensional patterns that can help investigation and root cause analysis of anomalous behavior in your data. As it analyzes the differences between 2 sets of records, it's more complex to use compared to autocluster or basket plugins, but is more powerful.

Also note that while in this example we generated differences for different points in time, we can use other dimensions besides time to split our data. For example, we can analyze http requests over one week and try to characterize the attributes of slow requests vs fast requests. In this example the duration of the request could be used to split the records set to 2 subsets.

We might specify a duration of less than one second for our baseline subset, and durations that exceed 10 seconds for our anomaly subset.

Module 4 - Time Series Analysis 1 - Creation and Core Functions

Demo 1 - Time Series Creation

Overview

In this demo we'll see how to create a *time series*. A time series is a dataset designed to be analyzed over time.

The KQL `make_series` operator plays a key role in the creation of a time series. It works by taking an input dataset, and aggregating a value based on time, to create a new dataset.

Examining the Code

The data we will be using for this demo comes from the `demo_make_series1` table, part of the public sample database from Azure Data Explorer. Let's get a quick look at the source data. We'll use the following query to bring back a few rows of data.

```
demo_make_series1
  | take 5
```

Below is a sample of the data. Note that your results may be different, the `take` operator grabs a random set of rows which can be different upon each execution of the query.

TimeStamp	BrowserVer	OsVer	Country
2016-08-25 09:12:58.0110000	Chrome 52.0	Windows 10	United Kingdom
2016-08-25 09:12:58.1470000	Internet Explorer 11.0	Windows 10	United Kingdom
2016-08-25 09:12:58.7420000	Firefox 48.0	Windows 8.1	Netherlands
2016-08-25 09:12:59.2140000	Edge 12.10240	Windows 10	Netherlands
2016-08-25 09:13:01.7480000	Chrome 52.0	Windows 7	India

When someone accesses our system, we log the date / time they did so, along with the browser they were using, what operating system, and what country they came from.

Our goal for this analysis is to easily compare number of logins by the users operating systems. Let's look at the query needed to accomplish this.

We'll start by declaring two variables.

```
let min_t = toscalar(demo_make_series1 | summarize min(TimeStamp));
let max_t = toscalar(demo_make_series1 | summarize max(TimeStamp));
```

These should be familiar to you by now. We are simply taking our dataset, `demo_make_series1` and getting the minimum and maximum date/time, then saving them into the variables.

Next, we'll use `make-series` to aggregate our counts and create a new dataset.

```
demo_make_series1
  | make-series NumberOfEvents=count() default=0
               on TimeStamp
             from min_t to max_t step 1h
               by OsVer
```

In the call to `make-series`, we first indicate we want to aggregate based on count, counting the number of rows. We are naming the count `NumberOfEvents` so in the output dataset it will have a meaningful name. We also supply a value to use as the default value in case there are no rows in specific time bins. For analyzing time series it's very important to keep all time bins, filling empty ones, otherwise we distort the time axis which can lead to faulty analysis.

Now we indicate what axis to aggregate on, in this case it is our `TimeStamp` column.

In the `from` line, we first set the range for our query, here going from the minimum time to the maximum time in our dataset.

The next part of the line, `step`, is important. Here we use a step of 1 hour. This means our counts will be aggregated into one hour buckets, also called bins.

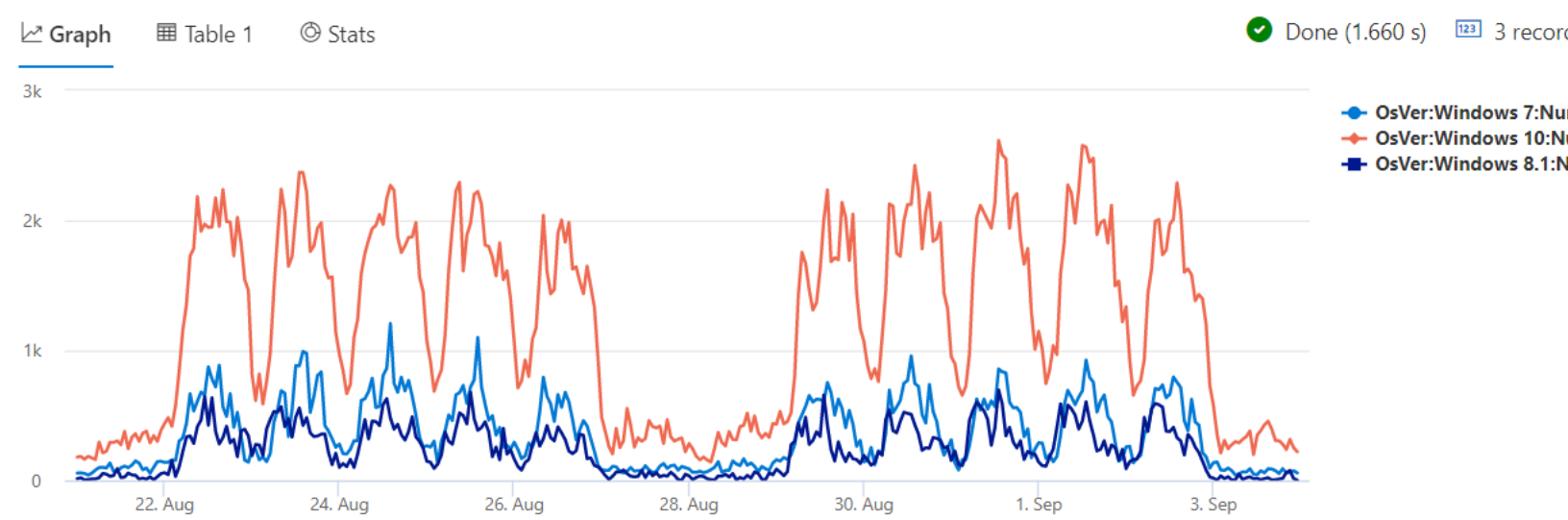
We add one more component, the `by OsVer`. The `OsVer` will be the partition key. We will aggregate and create a separate time series for each OsVer. There's still one more line to add to our query.

```
  | render timechart
```

This, as you've seen many times now, renders our output into a nice chart.

Analyzing the Output

And here is that chart.



When rendered like this, it's extremely easy to see that logins from Windows 10 computers far exceed those of older versions of Windows. You can also see the seasonal patterns over the 2 weeks of data. We can easily see the weekly seasonality - weekdays activity that is much higher than the weekend (in the middle of the chart), and the daily seasonality – midday activity is much higher than the night one.

Summary

The `make-series` operator is a foundational component to time series analysis. We will build upon it, adding additional time series functions in the next demo.

Module 4 - Time Series Analysis 1 - Creation and Core Functions

Demo 2 - Time Series Analysis Functions

1.1 Filtering Overview

This demo will comprise a number of functions which focus on time series analysis. The first of these is `series_fir`. FIR stands for Finite Impulse Response, and is used to calculate moving averages, change detection, and more. FIR is used in almost all signal processing systems including radio or video broadcasting, speech and sound processing and many other domains.

1.1.1 Examining the Code

If this first example looks familiar, it should. It is the same query with we ended the previous demo, with one addition.

```
let min_t = toscalar(demo_make_series1 | summarize min(TimeStamp));
let max_t = toscalar(demo_make_series1 | summarize max(TimeStamp));
demo_make_series1
  | make-series NumberOfEvents=count() default=0
              on TimeStamp
            from min_t to max_t step 1h
              by OsVer
  | extend ma_num=series_fir(NumberOfEvents, repeat(1, 5), true, true)
  | render timechart
```

On the next to the last line we've added an `extend` to add a new column. We're naming the new column `ma_num`, which stands for *moving average number*. It is the `series_fir` that will calculate it for us. Let's look at the various parameters.

The first one, `NumberOfEvents`, is the column containing value we want to average.

In the next position, we have the filter. Technically, this contains the coefficients of the filter, but you can think of it as defining the shape of the block for the moving average. If we wanted to create a moving average of five values, we could define the filter as [1,1,1,1,1]. We can also utilize the `repeat` operator, here we repeat the number 1 five times, which produces the same result.

The third parameter indicates whether we want to normalize our results. Normalization ensures the sum of the coefficients is 1, so we neither amplify nor attenuate the original signal. In our example it will transform the vector of coefficients from [1, 1, 1, 1, 1] to [0.2, 0.2, 0.2, 0.2, 0.2]. Note that if there are any negative coefficients we cannot normalize and this value must be set to false. It is an optional parameter, so if omitted it will default to true, unless there are negative values in the filter, in which case it will default to false.

The last parameter indicates whether we want to do a centered average or not, with false being the default. It will be easier to explain this using an example.

Let's say the first five values in NumberOfEvents is 10, 20, 30, 40, and 50. If we use a centered average, it will take the current value and center it in the middle of our [1,1,1,1,1] filter. Because there are no values before 10, since it's first, our average would be calculated as 0, 0, 10, 20, 30, resulting in a moving average of 12. On the next pass through the data, the next value of 20 takes the center spot, and our values become 0, 10, 20, 30, 40 resulting in a moving average of 20.

Let's contrast this with a backwards looking average. In it, values roll into the last position first, thus the first time the average is based on 0, 0, 0, 0, 10, resulting in 2. When the next sample is processed, we have 0, 0, 0, 10, 20, resulting in 6.

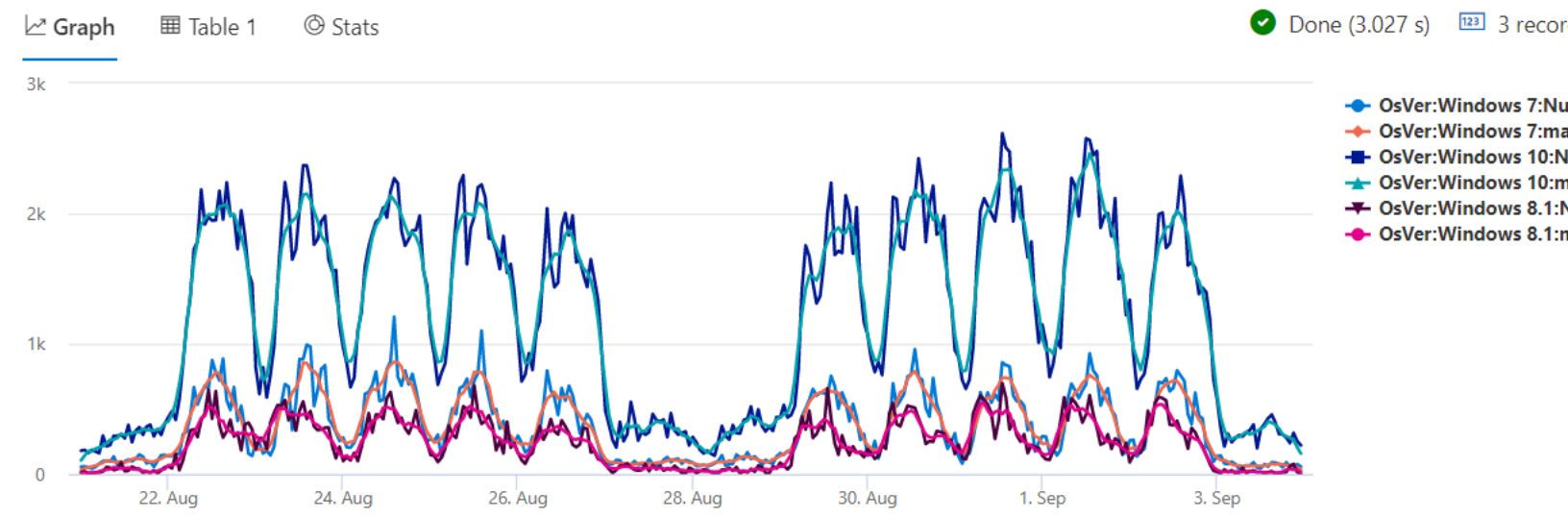
For streaming data we would like to set center to false, i.e. backward looking, as we don't have future data and we can only average from the current sample backward. But note that averaging backward introduce some time delay, i.e. the moving average lags behind the original signal.

On the other hand, when we process historical data, then we prefer to set center to true. With historical data, every sample already has its "future" data, so we can perform symmetrical averaging around the specific sample, avoiding the time delay.

In our query, we've indicated we want to use normalization, as well as a centered moving average.

1.1.2 Analyzing the Output

Our query ended with the command to render it as a timechart, so here is that chart.



In this chart we can see for each of the time series the respective moving average which is much smoother, as the moving average filtered the small fluctuations. Note that because we set center to true, the peaks and troughs of the moving average and the original metric lines are aligned.Had we set center to false they will be lagged behind the original ones.

1.2 Regression Analysis Overview

We can use regression analysis to calculate the line that best fits a series. In this demo, we'll look at two functions that can do this for us, `series_fit_line` and `series_fit_2lines`.

The `series_fit_line` function performs a linear regression to calculate the best line's parameters as well as its values. The `series_fit_2lines` is similar but more advanced - it finds the optimal split point and fit 2 independent line segments, where the first one fits the series data from the start to the split point, and the second one fits the data from after the split point to the end.

1.2.1 Examining the Code

The code for this looks relatively simple.

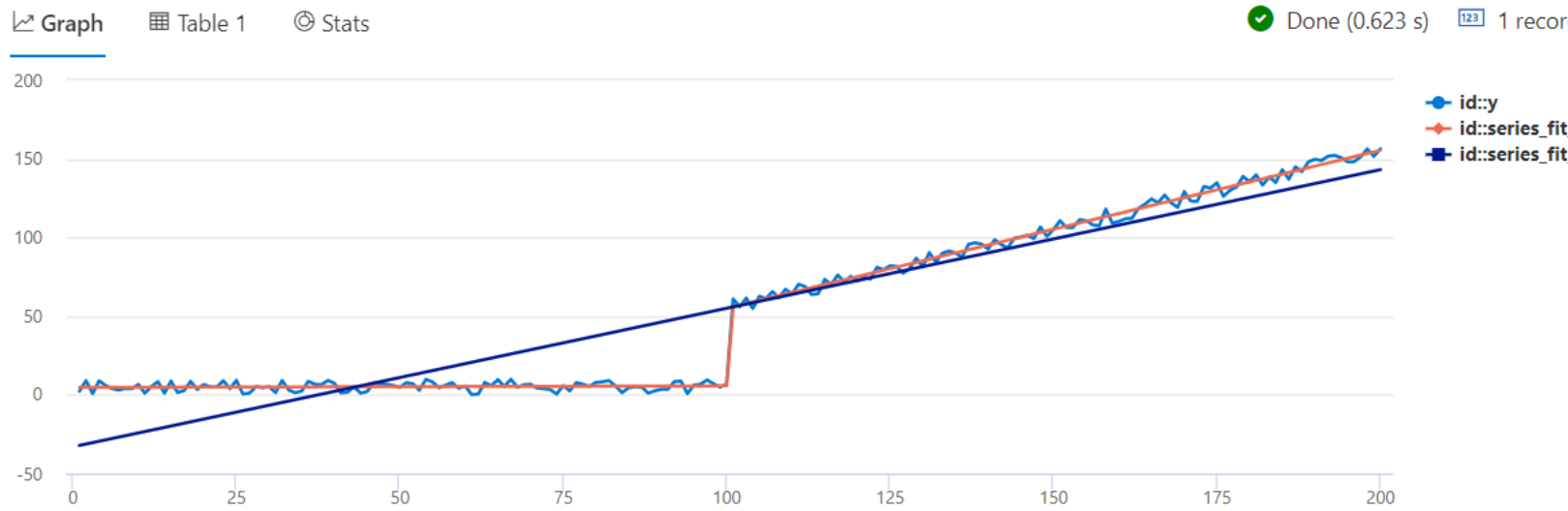
```
demo_series2
  | extend series_fit_2lines(y), series_fit_line(y)
  | render linechart with ( xcolumn=x )
```

The `demo_series2` table contains two columns, `x` and `y`. It then adds two new columns of data to our dataset, using the already mentioned `series_fit_line` and `series_fit_2lines` functions. Note that in this example the X-axis doesn't contain time points but numbers, so (x, y) represent generic series, not necessarily time series.

Finally, we render our output. As it's a generic series, we render a linechart instead of a timechart, and use the `with` clause to specify the `x` column in our source dataset as the X-axis in the chart.

1.2.2 Analyzing the Output

The results of our query produce an interesting chart.



We can see that a single line is not optimal, as our series has step jump in the middle. on the other hand, two lines fit finds the split point correctly, and the fit of the 2 segments is almost perfect, putting aside the noise on the original series. We can now press the table button to review the fit parameters in tabular format. Just for easy reviewing, in the next query we will project only the relevant fit parameters.

1.2.3 Examining the Code

Rather than charting the data, we can use this query to retrieve only the slope and r-square values of the two regression functions:

```
demo_series2
  | extend series_fit_2lines(y), series_fit_line(y)
  | project series_fit_line_y_slope
          , series_fit_line_y_rsquare
          , series_fit_2lines_y_left_slope
          , series_fit_2lines_y_right_slope
          , series_fit_2lines_y_rsquare
```

1.2.4 Analyzing the Output

Here are the results of our query:

series_fit_line_y_slope	series_fit_line_y_rsquare	series_fit_2lines_y_left_slope	series_fit_2lines_y_right_slope	series_fit_2lines_y_rsquare
0.879936864393257	0.874944924804405	0.0100560431915836	1.00181379415772	0.997072277363839

First, let's look on the slope - for a single line it's 0.88, positive trend, where each step the values increase. For 2 lines we see that the left part is 0.01, almost constant, while the right part, after the split point has a clear positive trend of 0.997. These parameters match what we visually saw in the graphs.

R-square is a number between 0 and 1, representing the quality of the fit. If the r-square of 2 lines fit (`series_fit_2lines_y_rsquare`) is much better (higher) than r-square of a single line fit (`series_fit_line_y_rsquare`) it means that there is a true change point in the time series, that justifies breaking it to two distinct segments

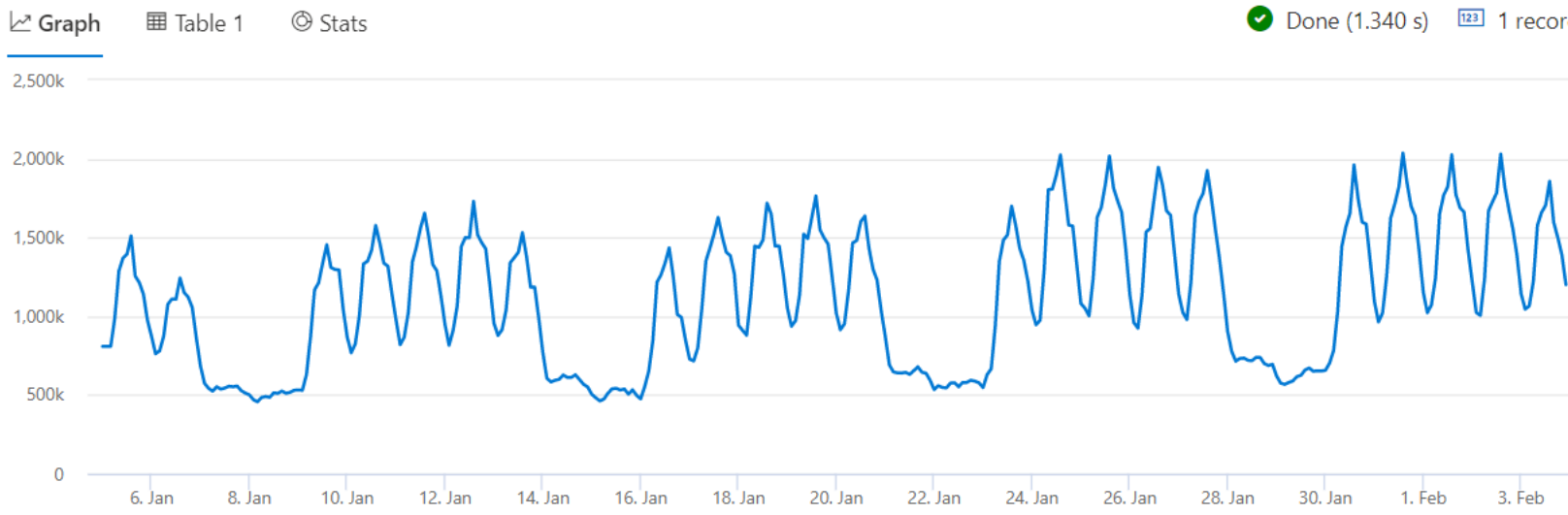
In this example, the R-square for fitting a single line is 0.8749, while the R-square for fitting 2 lines is 0.9970, which is almost perfect fit (without the noise R-square would be 1), just as we saw previously on the chart. Based on this output we indeed have a change point, justifying breaking our sample into two distinct segments.

1.3 Seasonality Detection Overview

Seasonality detection allows us to detect repeating patterns in our series, if exist. For example, in users' login activity we expect to have daily patterns, with peak activity in midday and minimal activity at night, as people are sleeping.

1.3.1 Examining the Code

For this demo, we'll be using `demo_series3` table containing two columns: `t` which is a times vector and `num` which is an activity vector. Note that this is a "pre-cooked" table that was the output of running make-series operation on the original table, using 2 hours step. Before we look at our main query, let's visualize the data, using the query `demo_series3 | render timechart`.



Here you can see the data displays 2 seasonal patterns - one is the weekly pattern, where activity is high during the weekdays and low during weekends, and the other is the daily pattern, where activity is high during midday and low during night. Now we'd like to automatically detect these patterns. Here's the query to find it:

```
demo_series3
  | project (periods, scores) = series_periods_detect(num, 0., 8d/2h, 2)
```

We start of course with our dataset, piping it into the function `series_periods_detect`. In the first parameter we pass in the column name we want to analyze, in this case `num`. In the next two columns we pass in the minimum and maximum periods to search for. This is just a hint for the algorithm, to limit the periods search range. In this case, based on out domain knowledge, we do not expect to have periods longer than a week, so we limit our search to 8 days, taking some spare beyond 1 week.

Note that each of the time series values in `num` represent count in 2 hours bins, so we divide 8 days by 2 hours to get the search range in pure bins units. In our case, we look for periods whose length is up to 8d/2h=96 bins. Finally we tell the function the maximum number of periods to return, in this case we want 2.

The `series_periods_detect` returns two columns, `periods` and `scores`, so we'll use project to make these the main output of our dataset.

The two columns are actually arrays, so we'll need to use `mv-expand` to convert them into individual rows.

```
  | mv-expand periods, scores
```

With that done, we can now add a new column to indicate the number of days for our periods.

```
  | extend days=2h*todouble(periods)/1d
```

Remember, each period is 2 hours. So first, we will multiply the periods by two hours to get a total value in hours. We then take that result and divide by 1d to convert the hours into days. That calculation is then placed in the new column, `days`.

1.3.2 Analyzing the Output

Here is the results of our query.

periods	scores	days
84	0.820622786055595	7
12	0.764601405803502	1

We see the number of periods, as well as the number of days represented by the periods. In our case, as we saw in the timechart, the function identified correctly both the weekly and the daily patterns. The score (a number between 0 and 1) indicates the confidence of that period. In our example the weekly score (0.82) is bigger than the daily score (0.76). This is due to the fact that the weekly pattern repeats consistently over the 4 weeks of our data, while the daily pattern repeats for the 5 weekdays but than changed for the 2 weekend days.

1.4 Element-wise Functions Overview

Our last demo in this section will build upon the first demo. In it, we used the `series_fir` function to calculate a moving average. We then plotted the moving average as well as the original value on a chart.

It would be nice to have the ability to highlight the difference in those two series. It sure would be nice if there was a function that would iterate over each element in the series and calculate it for us. Oh wait, there is! `series_subtract` is the solution here.

1.4.1 Examining the Code

Our demo builds on the first one, with the addition of 2 lines of code:

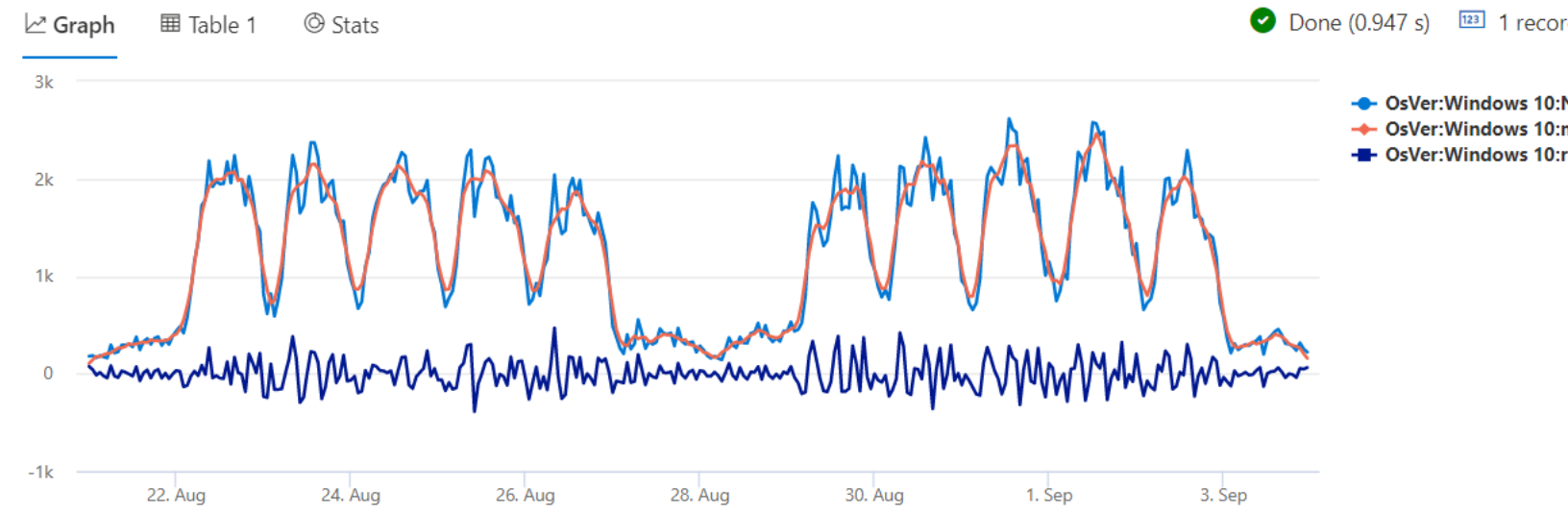
```
let min_t = toscalar(demo_make_series1 | summarize min(TimeStamp));
let max_t = toscalar(demo_make_series1 | summarize max(TimeStamp));
demo_make_series1
  | make-series NumberOfEvents=count() default=0
              on TimeStamp
            from min_t to max_t step 1h
              by OsVer
  | extend ma_num=series_fir(NumberOfEvents, repeat(1, 5), true, true)
   //to calculate residual time series
  | extend residual_num=series_subtract(NumberOfEvents, ma_num)
  | where OsVer == "Windows 10"
  | render timechart
```

Here we added a new line, in which we call the `series_subtract` function. We pass in the two series we want to subtract. In this case the second value, here `ma_num`, is subtracted from `NumberOfEvents`, the original series.

We added one other line, a where clause to limit us to just Windows 10. This simply made the new timechart easier to read for demo purposes.

1.4.2 Analyzing the Output

Our timechart now shows the addition of a third line.



The bottom line represents the difference in the two values. Having it plotted as a separate line makes it easy to see and identify the variations that were filtered by the moving average.

Note there are other similar functions, such as `series_add`, `series_multiply`, `series_divide`, `series_greater`, `series_less`, and many more.

Module 4 - Time Series Analysis 1 - Creation and Core Functions

Demo 3 - Time Series Workflow at Scale

Overview

In this final set of demos, we want to do two things. First, we want to show how well Azure Data Explorer process large numbers of time series. Second, we want to create a query that builds on the things you've already learned in this model to look for top anomalous time series out of thousands of time series based on the results of a linear regression analysis.

Examining the Code

To start, lets run a few queries to get an idea of the scale of the data we're working with. In this demo, we'll use the `demo_many_series1` as our data source. Running a simple `demo_many_series1 | count` lets us know there are (at the time of this recording) 2,177,473 rows.

We can get an idea of the data contents by grabbing the first few rows with `demo_many_series1 | take 4`.

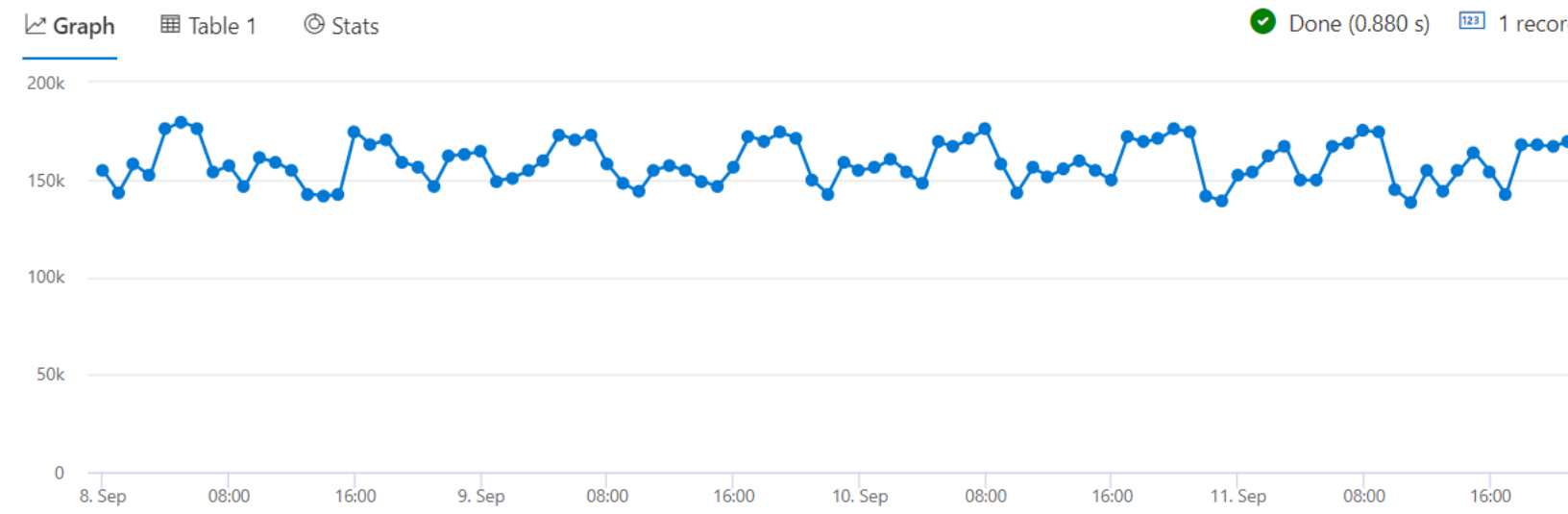
TIMESTAMP	Loc	Op	DB	DataRead
2016-09-11 16:00:00.0000000	Loc 11	75	528	277534
2016-09-11 16:00:00.0000000	Loc 11	37	542	193086
2016-09-11 16:00:00.0000000	Loc 11	78	542	791
2016-09-11 16:00:00.0000000	Loc 11	81	542	791

In addition to our timestamp, we have the location, Ops, DBs, and amount of data read. This table represents read counts of a distributed service over 4 days way back in 2016.The combination of Loc, Op and DB represent a unique instance of the service.

Now we'd like to identify anomalous behaviors in the data. At first, you might be tempted to run this query:

```
let min_t = toscalar(demo_many_series1 | summarize min(TIMESTAMP));
let max_t = toscalar(demo_many_series1 | summarize max(TIMESTAMP));
demo_many_series1
  | make-series reads=avg(DataRead)
              on TIMESTAMP
            from min_t to max_t step 1h
  | render timechart with (ymin=0)
```

And this will yield output.



This read count timechart present the avg read count over 4 days over all instances of the service. It looks stable and healthy, there are some minor fluctuations in the read count but that's normal. But is it indeed normal for each instance of the service? Let's check...

Let's take just a moment to get an idea of the scale of the data we are working with. As we'll be generating a time series for each combination of Loc, Op, and Db, this query will let us know how many instances we are working with.

```
demo_many_series1
  | summarize by Loc, Op, DB
  | count
```

This shows, at the time of this recording, we have 18,339 combinations of Loc, Op, and Db.

Now that we know the scale we are working with, let's proceed to a query that will analyze all 18,339 instances and let us know the ones we should be looking at.

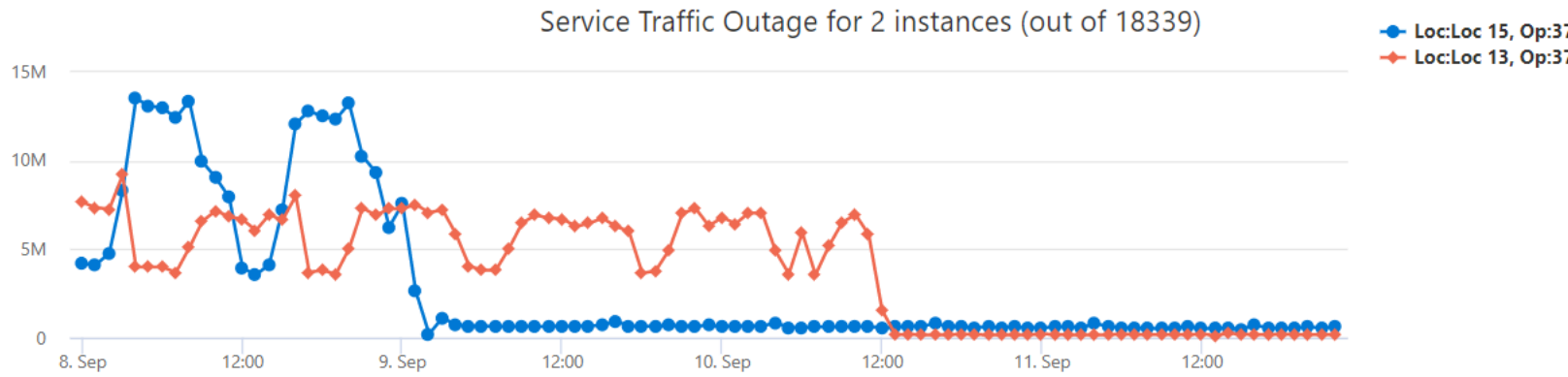
In the following query we create a time series for every instance of the service, that is characterized by the Loc,Op and DB triplets. There are more than 18,000 time series. This is done by adding the `by Loc, Op, DB` to the end of the `make-series` command. Then we run linear regression on all time series at once, and select the top 2 that had the most negative trend. Finally we render them.

```
let min_t = toscalar(demo_many_series1 | summarize min(TIMESTAMP));
let max_t = toscalar(demo_many_series1 | summarize max(TIMESTAMP));
demo_many_series1
```

```
  // Create a new series from our data aggregated by the Loc, Op, and DB
| make-series reads=avg(DataRead)
          on TIMESTAMP
        from min_t to max_t step 1h
          by Loc, Op, DB
| extend (rsquare, slope) = series_fit_line(reads)
| top 2 by slope asc
| render timechart
    with (title='Service Traffic Outage for 2 instances (out of 18339)')
```

Graph | Table 1 | Stats | Done (2.446 s) | 2 recor

Service Traffic Outage for 2 instances (out of 18339)



We can see indeed that unlike the single chart (which aggregated all 18000 service instances) that looked normal, using bulk linear regression we extracted 2 specific instances of the service that stopped working, as their read count was normal but dropped to almost zero

Summary

In the final demo of this module, we built on what we've learned. We have analyzed thousands of time series in few seconds and extracted few "needles", problematic instances of the service, out of the "big haystack" of all instances. This usage pattern is very powerful for real time health monitoring of cloud services, IoT devices and more.

Module 5 - Time Series Analysis 2 - Anomaly Detection and Forecasting

Demo 1 - Seasonal Decomposition

Overview

Decomposition allows us to take a time series and break it down into multiple components. These components include seasonal, trend, residual, as well as the baseline.

The baseline is the predicted value, to be used for forecasting. The residual is the input value minus the baseline, to be used for anomaly detection.

Examining the Code

The source dataset for this demo is `demo_make_series2`, which has three columns. The first is `sid`, and is simply a server ID. Next is `TimeStamp`, the date/time the activity was logged. Finally is `num`, the sum of traffic on our server.

Let's break down our query into smaller components.

```
let min_t = datetime(2017-01-05);
let max_t = datetime(2017-02-03 22:00);
let dt = 2h;
```

We start as we often do, with some variable declarations. The first two mark the start and end date ranges for this query. The last we'll use for the bin value, breaking our data into two hour buckets.

Next we employ our trusty pal `make-series` to summarize the data.

```
demo_make_series2
  | make-series Traffic=avg(num)
               on TimeStamp
             from min_t to max_t step dt
               by sid
```

We'll aggregate our traffic using average, and rename it to something that is more descriptive. Next we add the TimeStamp to indicate the axis to aggregate on. The `from` line is used to define the aggregation boundaries and the step size in two hour buckets. Finally we'll add the `by` clause to create individual time series per server.

To make our output easier to read, we'll now add a filter for a single time series:

```
  | where sid == 'TS1'
```

Next we call `series_decompose`

```
  | extend (baseline, seasonal, trend, residual) = series_decompose(Traffic, -1, 'linefit')
```

First, as indicated in our opener, the `series_decompose` returns four columns, all of them are time series components, so we'll need to use extend to add these columns to the time series created by `make-series`. In this example we explicitly specify (baseline, seasonal, trend, residual), a 4 element tuple to name the returned components, but we could omit it, using

```
  | extend series_decompose(Traffic, -1, 'linefit')
```

and the function will return default names for the new columns.

The first parameter we pass into `series_decompose` is the name of the column to be analyzed, in this case `Traffic`.

The next parameter defines how to calculate the seasonal component. A value of `-1` tells our function to autodetect seasonality, behind the scenes it uses the function `series_periods_detect()`, which we already learned about. We can also provide a specific value, for example 84 (the number of 2 hours bins in a week), in that case we force using weekly seasonality, or `0`, which tells `series_decompose` to assume there is no seasonality.

The third parameter indicates how we want to calculate the trend component. Here, we are passing in the value of `'linefit'`, indicating we want linear regression, by using the function `series_fit_line()` that we already learned too. Note that the default is `'avg'`, which as it implies, will use average value of all bins for the trend component. Usually linear regression is needed for long periods, few weeks and beyond, while it might be redundant for short periods. We can also pass `'none'`, in which case `series_decompose` assumes there is no trend.

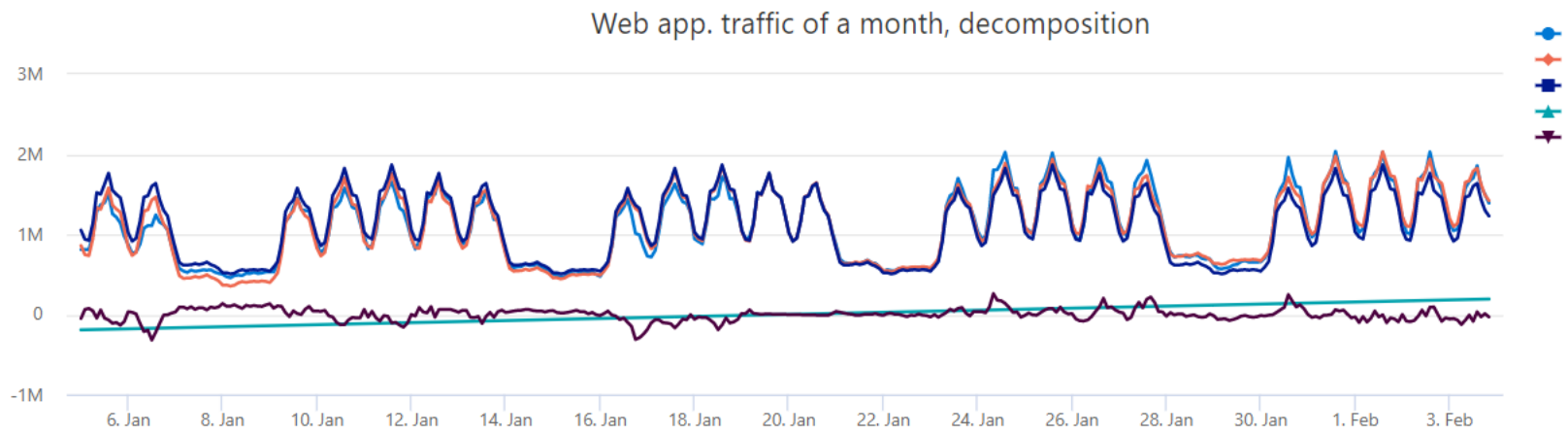
Finally, we'll tell the query to render the results into a timechart.

```
  | render timechart
      with (title='Web app. traffic of a month, decomposition')
```

Analyzing the Output

Graph | Table 1 | Stats | Done (2.758 s) | 1 recor

Web app. traffic of a month, decomposition



To better view you can hover or click on specific lines in the legend to highlight or view/hide them. Start by looking at `Traffic`, the original time series, then inspect `seasonal`, capturing the weekly pattern, `trend`, showing the gradual increase in traffic week over week and the `residual` which is the rest of the signal that we haven't modeled. Finally inspect `baseline`, the predicted traffic, which is just the sum of the seasonal and the trend components. In the next demos we will look for anomalies on the residual component, and forecast future traffic based on the baseline component.

Summary

Using seasonal decomposition, you can break your time series into multiple components, each represent specific characteristic of the series. You can then use these components for further analysis, including anomaly detection and forecasting.

Module 5 - Time Series Analysis 2 - Anomaly Detection and Forecasting

Demo 2 - Anomaly Detection

Overview

Anomaly detection allows us to identify outliers in our data. We can perform anomaly detection using the function `series_decompose_anomalies`. As its name implies, it builds on the `series_decompose` function we looked at in the previous demo. As many of the parameters function identically, we'll refer you back to the previous demo for in-depth details on these.

Examining the Code

The first part of the query is identical to the previous demo for `series_decompose`.

```
let min_t = datetime(2017-01-05);
let max_t = datetime(2017-02-03 22:00);
let dt = 2h;
demo_make_series2
  | make-series Traffic=avg(num)
            on TimeStamp
          from min_t to max_t step dt
            by sid
    // select a single time series for a cleaner visualization
  | where sid == 'TS1'
```

We declare a time range, two hour step value, pipe the data into `make-series`, and step over the data in two hour increments by our server ID. Finally we filter down to just a single server.

Now we can call the `series_decompose_anomalies` function.

```
| extend (anomalies, score, baseline) = series_decompose_anomalies(Traffic, 1.5, -1, 'linefit')
```

The first parameter is the column name with the value to analyze. The second parameter indicates the threshold for detecting anomalies.This should be a positive value, the smaller the value, the higher detection sensitivity. The default is 1.5 which detect mild anomalies, set it to 3.0 to detect strong anomalies.

The next two are seasonality and trend, which work identically to `series_decompose`. The default `-1` will auto detect seasonality, and `'linelift'` will use linear regression for the trend.

The function returns three columns, anomalies, score, and baseline, each being a time series. Let's review them from the last to the first one.

The 'baseline' is exactly the same component that we got from `series_decompose`, the predicted value of the Traffic.

The 'score' is a value representing how much each point deviates from the baseline. 0 means this points is identical to the prediction, positive value means it's above the prediction, the higher the value the bigger the deviation from the predicted value. Negative value means, similarly, it's below the prediction.

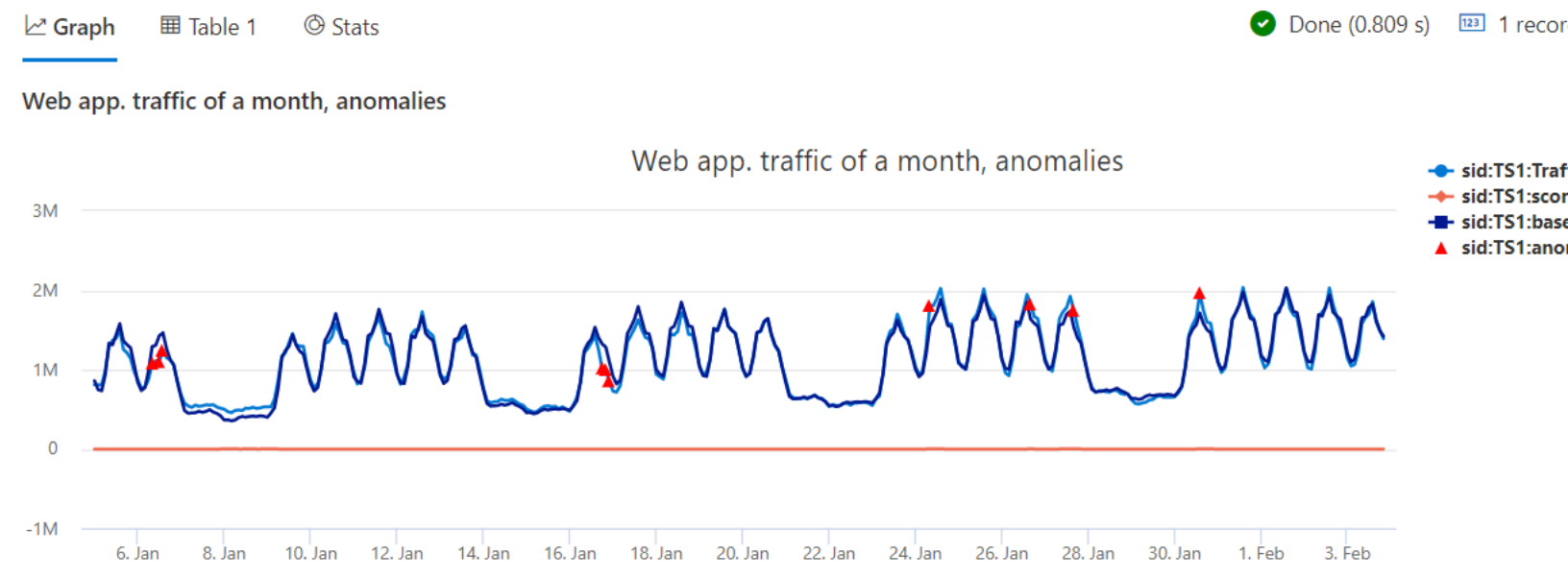
The last and most important columns is 'anomaly'. This is a ternary column, indicating for each point +1 if it's an high anomaly, -1 if it's a low anomaly and 0 if it's not an anomaly. This column is calculated just by comparing the 'score' to the anomaly threshold (the 2nd input parameter). If the *absolute* value of the score is below the anomaly threshold, then we set it to 0, as there is no anomaly. Otherwise, if the score is positive we set it to +1, as it's an high anomaly, and if the score is negative we set it to -1, as it's a low anomaly.

We'll then pipe this into a new type of chart, an anomaly chart.

```
| render anomalychart
    with (anomalycolumns=anomalies, title='Web app. traffic of a month, anomalies')
```

This is similar to a regular timechart, but specifying the anomaly column, it renders the anomalies as triangular bullets on top of the analyzed metric.

Analyzing the Output



In this chart, the anomalies appear as red triangles on the Traffic line. You can see that indeed the anomalous points deviate from the baseline. You can hover on one to get more details.

Summary

In this demo we presented how can we use the seasonal decomposition model for time series anomaly detection. Using this model is very powerful for detecting issues on top of expected metric that can have trend and seasonal patterns.

Module 5 - Time Series Analysis 2 - Anomaly Detection and Forecasting

Demo 3 - Forecasting

Overview

Another function in the seasonal decomposition family is `series_decompose_forecast`. Using it we can predict values into the future.

Examining the Code

The code for this demo is very similar to what we've seen so far.

```
let min_t = datetime(2017-01-05);
let max_t = datetime(2017-02-03 22:00);
let dt = 2h;
let horizon=7d;
demo_make_series2
  | make-series Traffic=avg(num)
            on TimeStamp
          from min_t to max_t+horizon step dt
            by sid
    // select a single time series for a cleaner visualization
  | where sid == 'TS1'
  | extend forecast = series_decompose_forecast(Traffic, toint(horizon/dt))
  | render timechart
     with (title='Web app. traffic of a month, forecasting the next week by Time Series Decomposition')
```

As before, we're using `demo_make_series2` as our source. The first addition you see is the declaration of the `horizon` variable. This indicates the duration for the forecast, here we set it to 7 days.

The second thing is in the `make-series` statement, where we set the upper boundaty `to max_t+horizon`. We already extend the input time series to include future timepoints for our forecast. We don't have values for these points so by default they are filled with 0.

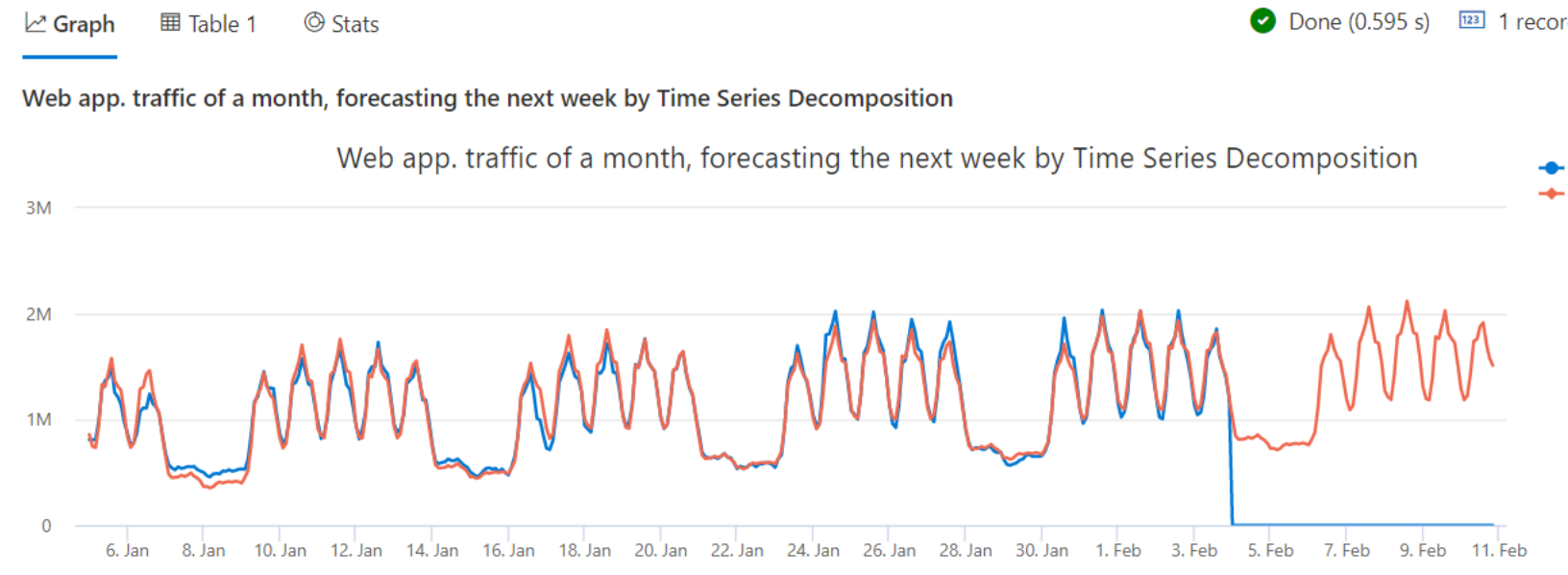
Then we call `series_decompose_forecast`. The first parameter is `Traffic`, the time series we want to forecast on.

The second is the number of points at the end of the time series that we want to exclude from the 'training', i.e. from building the seasonal decomposition model. As we extended the time series with zeros for the forecasting horizon of 7 days, we divide it by the 2 hours bin size. This yields the last 84 points to exclude from training. Note that we left these points off the training, but once the function builds the baseline component, it will extrpolate it to also forecast the 7 days horizon.

As the 2 other seasonal decomposition functions, this function can accept also the seasonality and trend parameters, but we didn't specify them explicitly, here we just left the defaults, -1 for auto detect seasonality and 'linefit' for trend analysis.

Finally we render the result as a timechart. Let's take a look.

Analyzing the Output



In this chart, we see two lines. For the past, you can compare the forecast (which is just the baseline that we have seen before) vs the actual Traffic. For the future of seven days, the forecast line predicts the Traffic values while we don't have real values for the actual traffic .

Summary

Using `series_decompose_forecast` you can predict future values of your time series. This function is also based on the seasonal decomposition model, where the baseline component is the prediction.

Module 5 - Time Series Analysis 2 - Anomaly Detection and Forecasting

Demo 4 - Scalability

Overview

In this final demo, we're going to combine what we've done in the previous demos of this module to produce a forecast for multiple time series.

Examining the Code

The first part of this query is identical to the previous demo.

```
let min_t = datetime(2017-01-05);
let max_t = datetime(2017-02-03 22:00);
let dt = 2h;
let horizon=7d;
demo_make_series2
  | make-series Traffic=avg(num)
              on TimeStamp
            from min_t to max_t+horizon step dt
            by sid
```

This time we do not filter as before for a single time series, `TS1`, but our pipeline shall process all time series, three in our demo, at once.

In the timechart, we are going to have six lines - 3 pairs, each containing the original time series and its prediction. This lines would be drawn one on top of the other, making it difficult to view. So for this demo we will separate them by adding artificial offset to each of the time series pairs.

```
  | extend offset=case(sid=='TS3', 4000000, sid=='TS2', 2000000, 0)
```

Here we are using the `case` statement. You can think of it as a series of if/then statements. Here, if the server id is TS3, it will place the value of 4,000,000 into our new `offset` column. If the server is TS2, 2,000,000 is put in `offset`. Finally, when case does not find another test condition but it does find a value, it interprets it as an `else` and will use the value of 0.

In an earlier demo we looked at the function `series_subtract`. Here we can use its peer, `series_add`, to add the value in the new offset column to the Traffic value.

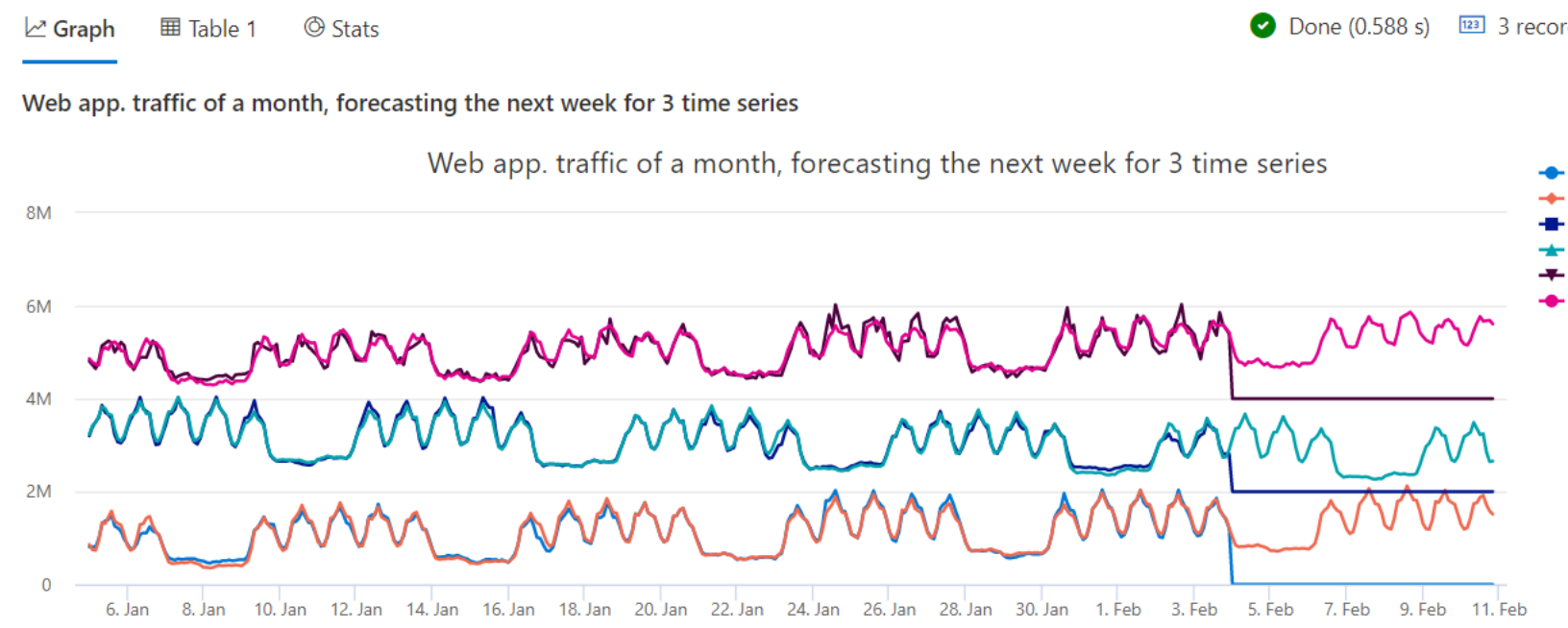
```
  | extend Traffic=series_add(Traffic, offset)
```

We wrap up by creating the forecast, then sending it to a timechart.

```
  | extend forecast = series_decompose_forecast(Traffic, toint(horizon/dt))
  | render timechart
    with (title='Web app. traffic of a month, forecasting the next week for 3 time series')
```

As we've seen these commands already we won't elaborate further, but go ahead and run the query.

Analyzing the Output



Here we can see both the traffic and forecast for all three servers. This information can help us in load balancing our servers in the future.

Summary

The goal of this demo was to present how to forecast multiple time series at once. You only need to create multiple time series using the `by` clause in `make-series`, then the syntax of the following processing pipleline is identical, regardless if it's a single time series or many.

Processing multiple time series is optimized and very fast, as it's implemented using vecotrization methods. Here the forecast was done for only three time series, but as you have seen in the last demo of the previous module, it can be done on thousands of time series in seconds. As said there, this usage pattern is very powerful for real time health monitoring as well as for resource planning of cloud services, IoT devices and other scenarios.

Module 6 - Extensibility Using inline Python and R

Demo 1 - Calling the Python Plugin

Prerequisites

In this demo we will see how to call both Python and R from within a KQL query. Due to security considerations, the Python and R plugins are not enabled in the Microsoft public cluster and samples database we've been using. As such, if you want to try the following demos, you will need to use your own cluster, within your own Azure environment. If you setup your own Dev cluster, the cost is very reasonable, at the time of this writing approximately $ 0.21 (yes, 21 cents in United States currency) per hour.

In addition, you will need to make sure that the python() and r() plugins are enabled in your development cluster, as it is disabled by default.

As this course focuses exclusively on the Kusto Query Language, setting up a cluster and enabling the Python/R plugins are outside the scope of this course. However Microsoft has provided some excellent guides.

First, there are several guides for using scripting languages to creating a cluster. This particular guide uses Azure Resource Manager templates.

Creating ADX Clusters and Databases Using ARM templates

In this same area are other How To guides using other languages, such as Azure CLI, PowerShell, Python, and more.

Once you've created your cluster, you can enable Python or R using the instructions here.

Manage Language Extensions in Your ADX Cluster

It is also assumed you are familiar with the Python and/or R languages, as we will not attempt to explain the Python code here. Pluralsight has many excellent courses on Python and R should you need a refresher course.

Finally, in order to keep this module consistent with the rest of the course, several of the tables in the Microsoft public samples have been cloned into the private cluster used for this particular module. You can easily clone the needed tables to your private cluster by running

```
.set tbl < cluster('help').database('Samples').tbl
```

from the context of your personal database, replacing `tbl` with the name of the required table to clone.

Overview

As awesome as the Kusto Query Language is, there are times when you need functionality not native to KQL. To enable this type of extensibility, the Azure Data Explorer team has created plugins for the Python and R languages, enabling you to embed Python/R code in your KQL query. We'll begin by seeing how to execute Python code from within a Kusto query.

A quick formatting note, when we use Python, with a capital P, we are referring to the Python language. When we use python(), with a lower case p and parenthesis, we are referring to the python plugin.

Examining the Code

The first thing to understand is that you cannot simply type in Python code into the Kusto query window, as I've done here. If I run this, it will result in a 'recognition error' which is essentially a syntax error. Instead, you have to place your code in a formatted multi-line string, then pass that string into the python plugin.

To help you out, there is a key combination you can use that will format your Python code into a string. Highlight the block of code you want to convert, and press Ctrl+K, then Ctrl+S.

You can now see my code has been wrapped in single quotation marks, making them a string. Additionally the line feed character of `\n` has been added to the end of each line.

Note this key combination works in both the ADX website as well as the Kusto desktop application.

OK, lets start looking deeper into the use of Python in your Kusto queries.

Here is a simple example.

```
range sourceNumber from 1 to 10 step 1
| evaluate python( typeof(*, powNumber:int)
                 , 'exp = kargs["exp"]\n'
                   'result = df\n'
                   'result["powNumber"] = df["sourceNumber"].pow(exp)\n'
                 , pack('exp', 4)
                 )
```

We'll explain this in a moment, but note that we have embedded the Python code directly into the query. As we said, you must format the Python code as a multi-line string by enclosing each line in quotes and end it with explicit newline: '...\n'. You can select the Python block in the query editor and use a keyboard shortcut Ctrl+K,Ctrl+S to automate this formatting for the full Python block.

To increase readability we can also put our Python code into a variable, and use that in our call.

```
let sourceDataset = range sourceNumber from 1 to 10 step 1;
let pyCode = 'exp = kargs["exp"]\n'
             'result = df\n'
             'result["powNumber"] = df["sourceNumber"].pow(exp)\n';
sourceDataset
| evaluate python( typeof(*, powNumber:int)
                 , pyCode
                 , pack('exp', 4)
                 )
```

As we've done in so many examples, we start by creating our source dataset. Here we've named it simply `sourceDataset`, and use the `range` command to generate a list of values, from 1 to 10, and store them in a column named `sourceNumber`.

Now let's talk about the mechanics. The Python code is run on a 'Python sandbox', a secure and isolated Python environment, based on Anaconda distribution, that is running on the same existing ADX compute nodes. We need to send to the sandbox the data set to work on, the Python script and also parameters for the script.

The data set is essentially an ADX table, that is sent to the sandbox and is mapped to a Pandas DataFrame which is named 'df', in the context of the Python environment.

The code is sent as formatted multi-line string as discussed above. The parameters are sent as a dictionary containing key/value pairs, which is mapped to a Python dictionary named 'kargs' in the Python environment.

Finally, the output of the Python script should be stored in another DataFrame, named 'result', that is sent back to ADX. So just like native operators, you can pipe a table to the python() plugin, and continue piping the resulting table to additional operators.

Back to our demo, reviewing the Python script, we can see that first we extract a parameter named `'exp'` from the dictionary of parameters. Then, we just copy the input DataFrame `df` to the output `result`. Finally we append a new column, `powNumber` which is calculated for each line by raising the number in the source column `sourceNumber` to the specific `exp` power.

With the dataset created, and Python code written, we now take our source, `sourceDataset` and pipe it to `evaluate` in order to call the `python` plugin.

The python plugin requires three parameters. The first is the output schema of the data. Note that ADX needs to know the output schema for every operator, to be able to perform the query planning phase (which optimize the query before its actual execution).

It should always use the `typeof` function to format it. In this example the first parameter to `typeof` is an asterisk `*`. This is a shortcut for the schema of the input table, in this case it's only `sourceNumber` of type `int`. Still, this shortcut is very handy for tables with dozens of columns, avoiding the need to explicitly type them one by one. Keeping all input columns is optional of course, you don't have to output whatever is input, but it's quite common that you just want to add few calculated columns.

In our example, in the second parameter to `typeof`, we create a new column name for our exponent output `powNumber` with its `int` type. This is the same column name that we appended in the `result` DataFrame in our Python script.

With the `typeof` being our first parameter, the second parameter is obviously our block of Python code. Storing it in a variable makes our query more readable, and much easier to work with and update later if needed.

The final parameter to the python plugin is dictionary of the key/value pairs (that was mapped to `kargs` in the Python environment). The KQL `pack` function will create a key/value dictionary. In our case it contains a single key named `exp`, whose value is 4, meaning that our Python script will raise the source numbers to the 4th power.

Analyzing the Output

The output is pretty much what you expect.

sourceNumber	powNumber
1	1
2	16
3	81
4	256
5	625
6	1296
7	2401
8	4096
9	6561
10	10000

Because we used the `*` as the first parameter in the `typeof`, our `sourceNumber` column from the source is present. In addition, our Python calculated column `powNumber` is also in the output.

Summary

The purpose of this demo was not to show how to create a complex calculation in Python, but rather to explain the 'python sandbox' environment and show the basic mechanics of calling the python() plugin. With this you should now have an understanding of:

- How the data from our dataset flows into the plugin
- How to declare the output of the plugin
- How to pass our Python code into the plugin
- And finally, how to pass parameters into the plugin

In future demos we will no longer elaborate on the mechanics of the plugin, nor will we dive into the composition of the supplied Python code. Instead we will be showing two popular use cases for combining KQL with Python, after which we'll do a quick demonstration of using R with ADX.

Module 6 - Extensibility Using inline Python and R

Demo 2 - Time Series Analysis with Python

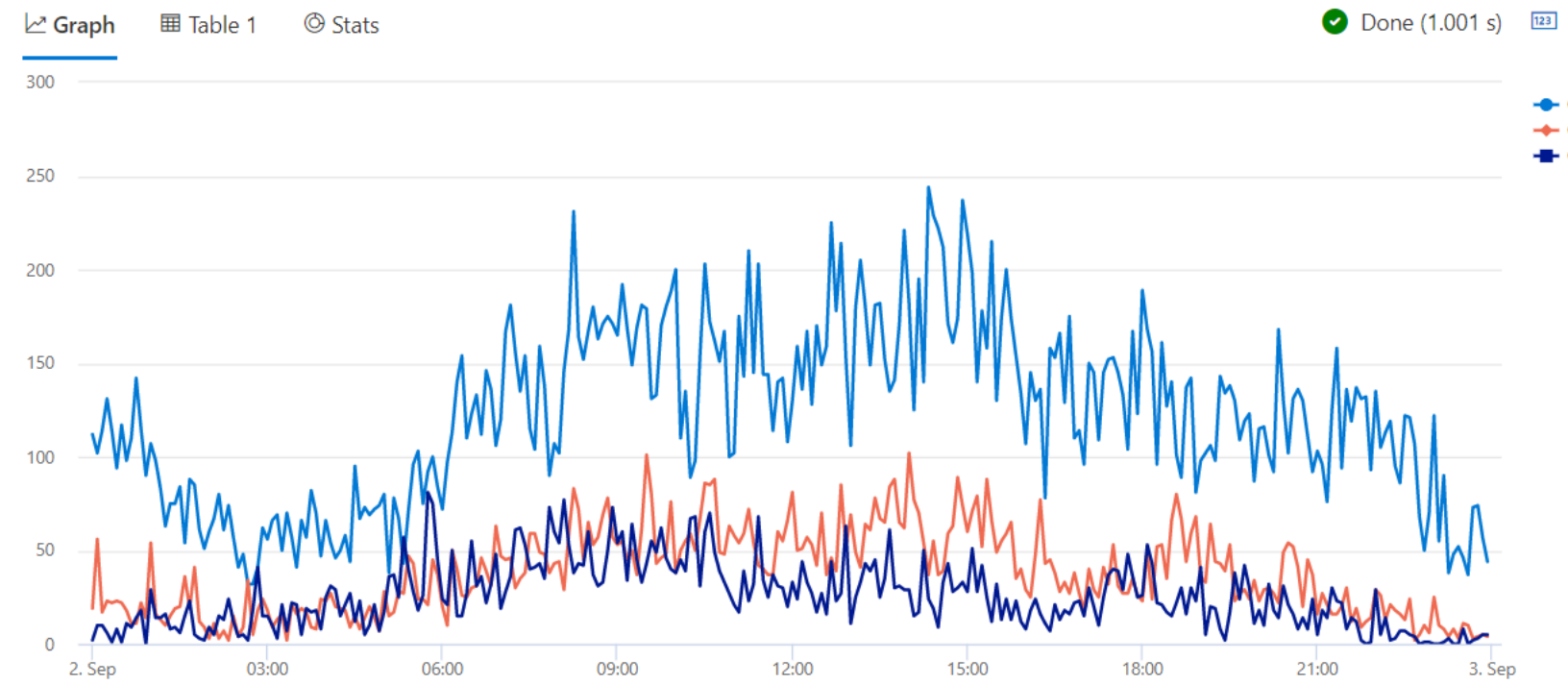
Overview

In this demo, we'll use the numpy package that is the major mathematical package of Python, in order to calculate best fit curves.

The Source Dataset

```
let max_t = datetime(2016-09-03);
demo_make_series1
  | make-series num=count()
              on TimeStamp
            from max_t-1d to max_t step 5m
              by OsVer
  | render timechart
```

This simple query is just a reminder of the dataset. We've seen queries very similar to this several times in recent modules. The chief difference is the time window, here we are selecting a single day and a bin of five minutes.



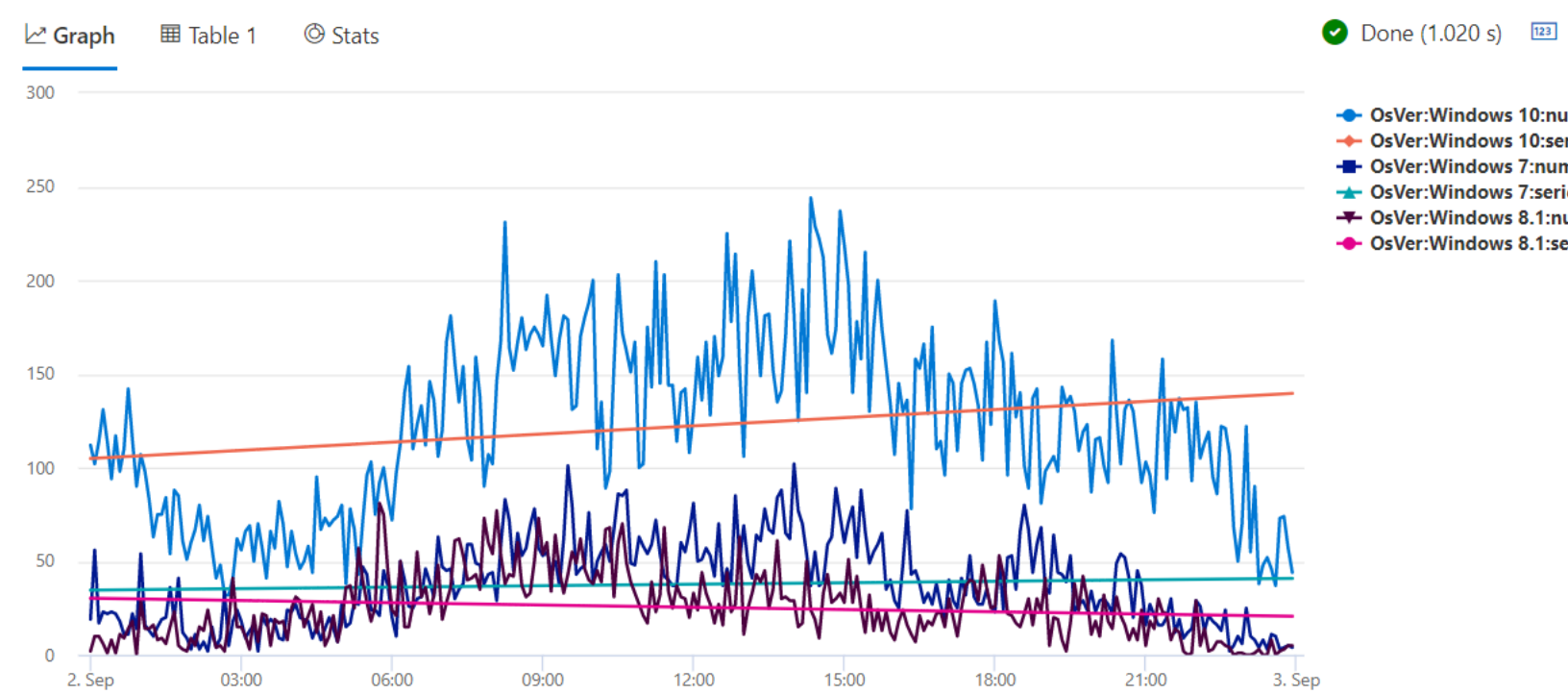
As you can see, this is simply the number of entries logged by each operating system over time, grouped in five minute bins.

Attempting to use Series Fit Line

As a logical first step to finding the best fit line for our data set, we'll use the built in Kusto function `series_fit_line` to perform our linear regression.

```
let max_t = datetime(2016-09-03);
demo_make_series1
  | make-series num=count()
              on TimeStamp
            from max_t-1d to max_t step 5m
              by OsVer
  | extend series_fit_line(num)
  | render timechart
```

As there are no new concepts here in terms of the KQL code, let's go ahead and run it to see the output.



Oh my! This isn't a good fit at all. Looking at any of the operating systems, we can see that the actual activity time series diverges wildly from the respective fitted lines.

So now what? Well, as you've probably guessed, we can use some Python to calculate a non-linear, high order polynomial curves, that fit the activity time series much better.

Using Python

We'll start the query by declaring a variable to hold our Python code.

```
let pyCode = 'in_series = kargs["in_series"]\n'
             'out_series = kargs["out_series"]\n'
             'degree = kargs["degree"]\n'
             '\n'
             'def fit(s, deg):\n'
             '# our fit function accept a single series\n'
             '# and the degree of the polynomial to fit\n'
                  //  create x axis with equally spaced sequential values
             '    x = np.arange(len(s))\n'
                  //  the best fit algebra is done here, returning the polynomial coefficients
             '    coeff = np.polyfit(x, s, deg)\n'
                  //  create a polynomial
             '    p = np.poly1d(coeff)\n'
                  //  extrapolate its values (y-axis) for the x values
             '    z = p(x)\n'
             '    return z\n'
             '\n'
             'result = df\n'
             //  use apply to call the fit function for each time series
             'result[out_series] = df[in_series].apply(fit, args=(degree,))\n';
```

As stated in the beginning of this module, we won't dissect the Python code here. We just added few inline comments, assuming that you have already experienced functions from the numpy package. The target of this demo is to explain how to embed this Python code inside KQL query.

I do want to call your attention to the comments. You can either use Kusto comments (marked by `//`) outside the strings that define the Python code, or Python comments (marked by '#') within the strings.

With the Python code variable created, we can now begin creating our query.

```
let max_t = datetime(2016-09-03);
demo_make_series1
  | make-series num=count()
              on TimeStamp
            from max_t-1d to max_t step 5m
              by OsVer
  | extend series_fit_line(num)
```

We continue from where we left off. Note we keep the `series_fit_line`, so we can compare it to the high order polynomial curves that will be generated in our Python code.

```
  | extend fit_num=dynamic(null)
```

When we call the python plugin, we will need a place to store the output time series. So here we define `fit_num` of `dynamic` type, to store the output time series arrays. Even though we set it in the Python code, we still need to initialize it, so we fill it with `null`s.

Now we're finally ready to call Python.

```
  | evaluate python( typeof(*)
                   , pyCode
```

```
, pack('in_series', 'num', 'out_series', 'fit_num', 'degree', 4)
)  //  passing dictionary of script parameters to the Python sandbox
```

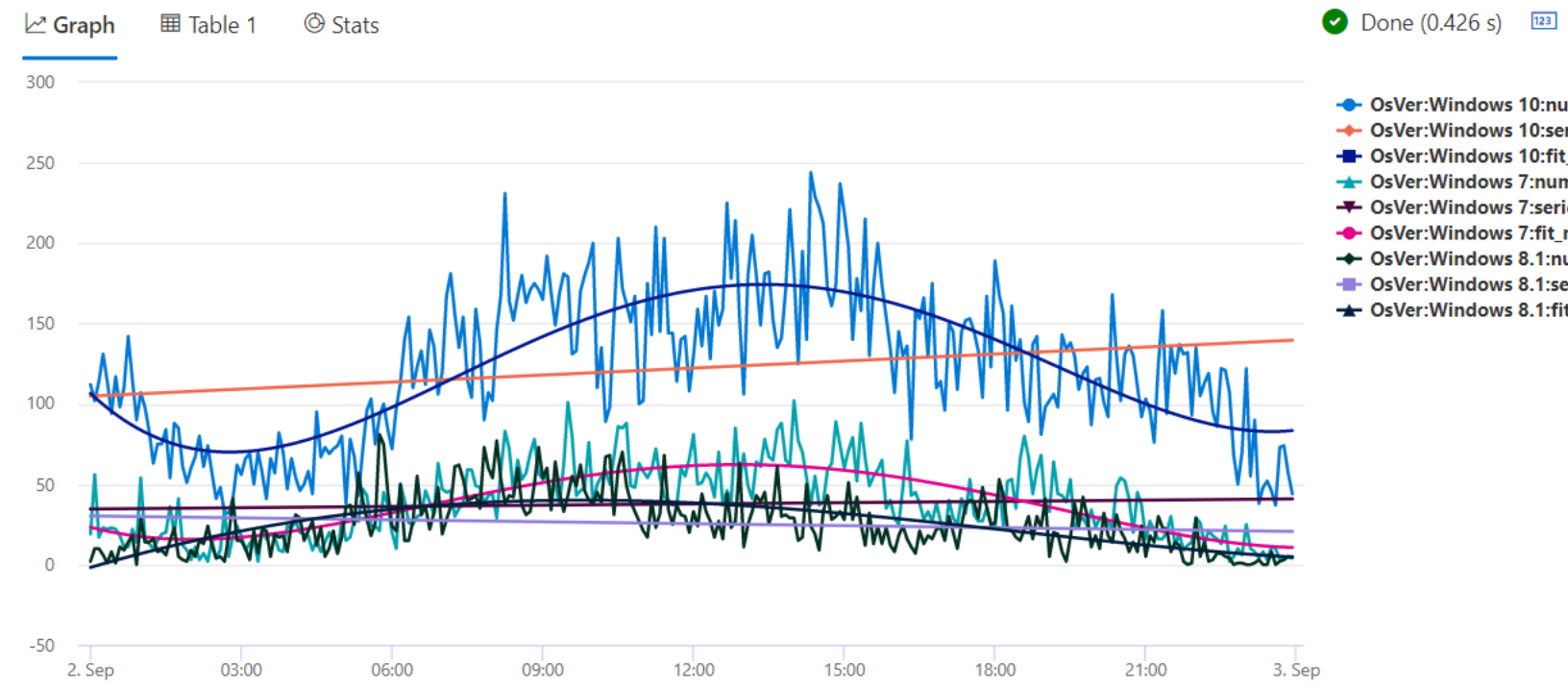
The first parameter, as you recall from the previous demo, specifies the output schema. Setting it to `typeof(*)` means that the output schema is the same as the input, i.e. all input columns are kept and there are no new columns (as we created the `fit_num` column before the call to python()).

The second parameter is the reference for the Python code. The last parameter is the key/value parameters dictionary. Here we pass `'num'`, the name of the column containing the input time series, `'fit_num'`, similar column name for the output time series and `4` as the degree for the polynomial fit (i.e. to fit by polynomials of the 4th order). Note that we pass the column names as strings, and the degree as int.

Finally we take all of this and render it as a timechart.

```
| render timechart
```

Let's run this to see the output.



As you can see, the dark blue curve of `Windows 10:fit_num` follows the path of the `Windows 10:num` value very closely, and is thus a far better fit than the built in `series_fit_line` that use linear regression (i.e. polynomial of 1st degree). Same for the other operating systems.

Summary

As flexible and powerful as the Kusto Query Language is, there are times when you have to step outside of it to analyze your data. Using the python() plugin you can leverage the rich and free Python eco-system from within your KQL query.

Be aware when using inline Python (or R), performance and scalability are inferior compared to using native KQL. The best practice is to first try and solve your case using native KQL operators and functions, saving Python (or R) for cases you cannot solve using ADX native capabilities.

Module 8 - Extensibility Using inline Python and R

Demo 3 - K-Means Clustering

Overview

In this demo, we'll clusterize a dataset by the well known K-Means algorithm. Using the data in StormEvents, we'll look at the start and end locations of the reported events and try to find distinct groups, each containing many events that occurred in approximately the same location.

In data science terminology, each group is called a cluster and is characterized by its size and centroid (the average location of each coordinate). Our KQL query will run K-Means and return the clusters' size and centroids.

You can find more information on K-Means Clustering at: K-Means Clustering on Wikipedia.

The K-Means algorithm is included in the scikit-learn package. scikit-learn is the main machine learning package on Python. For this demo, we are using sklearn.cluster.KMeans class.

Examining the Code

As is our standard, we'll begin by declaring the variable to hold our Python code.

```
let pyCode = 'from sklearn.cluster import KMeans\n'
             'k = kargs["k"]\n'
               //  instantiate the KMeans object for calculating k clusters
             'km = KMeans(n_clusters=k)\n'
               //  the actual clustering is done here
             'km.fit(df)\n'
               //  copy the centroids to the output dataframe
             'result = pd.DataFrame(km.cluster_centers_, columns=df.columns)\n'
               //  count the number of items in each cluster and copy it
             'result.insert(df.shape[1], "size", pd.DataFrame(km.labels_, columns=["n"]).groupby("n").size())\n'
               //  set sequential numbers for cluster id
             'result.insert(df.shape[1], "cluster_id", range(k))\n';
```

Briefly, the code imports the KMeans class from scikit-learn package, calculates our clusters and copies the clusters' centroids and size to the output dataframe.

Next is our dataset.

```
StormEvents
  | where isnotnull(BeginLat)
  | project BeginLat, BeginLon, EndLat, EndLon
```

Pretty simple stuff by now. We simply filter out the rows with a missing `BeginLat`, then reduce the output to only the four columns we need using `project`.

Next we call the python plugin.

```
  | evaluate python( typeof(*, cluster_id:int, size:long)
                   , pyCode
                   , pack('k', 5)
                   )
```

The call to the python plugin should be comfortable to you by now. We first declare the schema of the output. For storing the centroids we can just use the input schema that includes the beginning and ending longitude and latitude columns, as indicated by the `*`.

In addition, we'll need to extend it with two new columns. The first will store the cluster ID, the second will store the cluster size. We then pass in the Python code to execute, and finally the dictionary of parameters. Here it contains a single parameter, `k`, to indicate the number of clusters we want, and we set it to `5`.

Finally we just sort our output top down by cluster size:

```
  | order by size
```

Running this code, we get back the following output.

BeginLat	BeginLon	EndLat	EndLon	cluster_id	size
34.796871923447	-98.5346490537796	34.7977185715813	-98.52499762643	2	9361
42.578856478889	-93.2791174039646	42.5762450106395	-93.2708919363871	0	8926
33.4056244668625	-84.2177226024774	33.4042755331375	-84.2091818413996	3	7825
40.9578664281455	-77.3498180679785	40.9566293231962	-77.3425573196185	1	6708
41.6982654347061	-109.937147835269	41.7000902851109	-109.929269904963	4	2842

We can use these results for drawing the centroids on a map, each with its respective size or for any further analysis.

Summary

In the module Performing Diagnostic and Root Cause Analysis, we reviewed the built in clustering plugins including autocluster and basket. However, there are plenty of clustering algorithms. In this demo we have seen how to use Python's scikit-learn package for clustering by the well known K-Means algorithm. Using the python() plugin you can apply dozens of machine learning algorithms from a variety of Python packages including scikit-learn, TensorFlow, PyTorch and many more.

Module 6 - Extensibility Using inline Python and R

Demo 4 - Calling the R Plugin

Overview

As mentioned in the opening of this module, in addition to calling Python, ADX also has a plugin for the R language. Here we'll take a quick look at calling some R code.

Examining the Code

```
let rCode = 'result <- df\n'
            'n <- nrow(df)\n'
            'g <- kargs$gain\n'
            'f <- kargs$cycles\n'
            'result$fx <- g * sin(df$x / n * 2 * pi * f)';
range x from 1 to 360 step 1
    // Output schema: append a new fx column to original table
  | evaluate r( typeof(*, fx:double)
              , rCode
                //  dictionary of parameters
              , pack('gain', 100, 'cycles', 4)
              )
  | render linechart
```

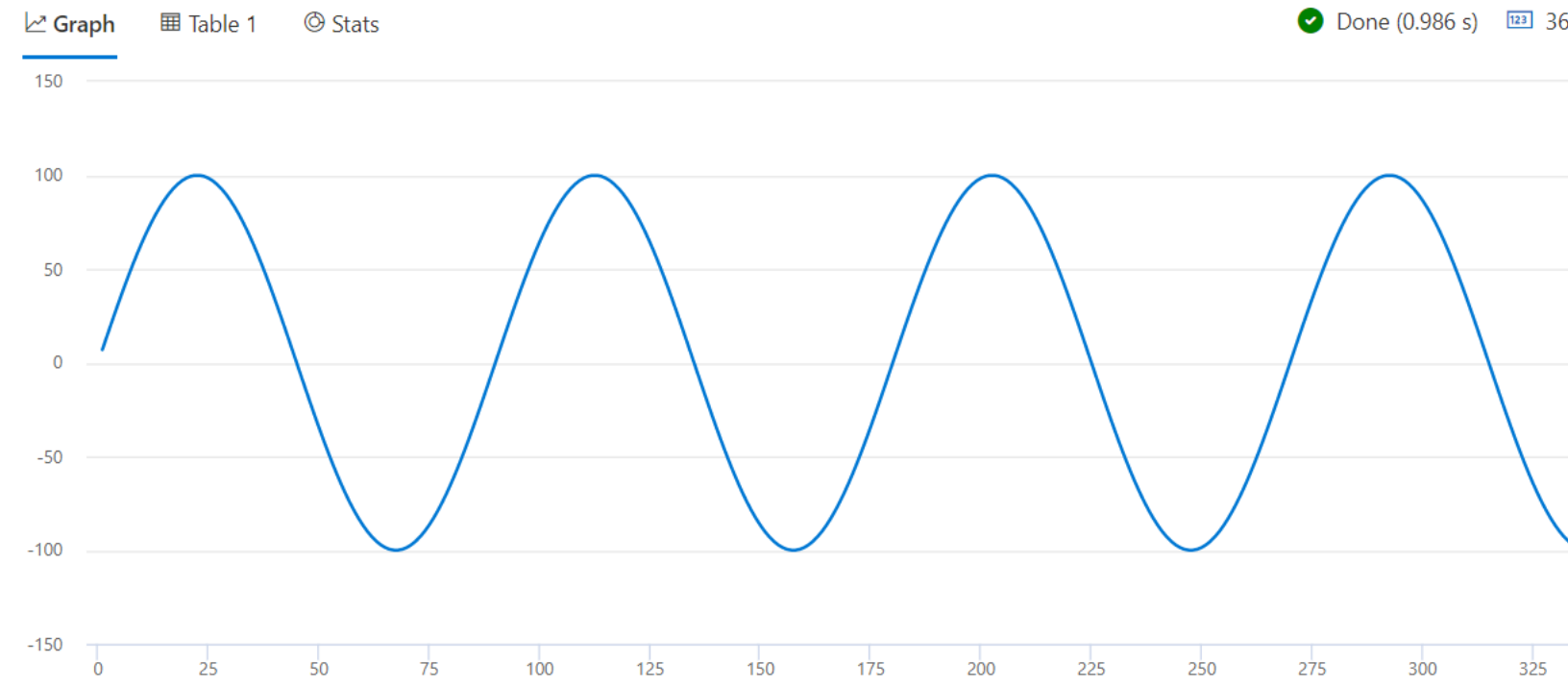
This code just calculates a simple sine wave using R and draws it.

As you can see the structure and mechanics are the same as for Python.The only difference is that instead of python() you call the r() plugin:

```
| evaluate r(...)
```

Analyzing the Output

Here is the linechart of our sine wave:



Summary

In this final demonstration, we learned how to call R from within KQL query. If it was not clear by now, you can build a single powerful KQL query, mixing native functions, Python and R, enabling you to enjoy the candies of all these great data science environments.

A final reminder, in order to perform the demonstrations in this one module, you will need to be running on your own private cluster with the Python and R extensions enabled.

If you setup a dev cluster for the purpose of testing out the code in this module, you should also remember to delete it when you no longer need it. While 21 cents (in US currency) per hour for a dev cluster is not a lot of money, it will begin to add up if you forget about it and leave it running.