

MUST LEARN

# KQL

THE SERIES

ROD TRENT  
SENIOR MICROSOFT SECURITY ADVOCATE  
MICROSOFT

*This is part of an ongoing series to educate about the simplicity and power of the Kusto Query Language (KQL). If you'd like the 90-second post-commercial recap that seems to be a standard part of every TV show these days...*

*The full series index (including code and queries) is located here:*

<https://aka.ms/MustLearnKQL>

*This book is updated every time a new part of this series is posted. The most current edition of this book will always be located at: <https://cda.ms/3m1>*

**Book release ver. 2.01, February 22, 2022 8:17am EST**

## Contents

<b>Must Learn KQL Part 1: Tools and Resources</b>	5
Reference	5
Practice Environments	6
Actual Books	6
Tools	6
Blogs, Websites, and Social	7
Video	7
GitHub Query Examples	8
<b>Must Learn KQL Part 2: Just Above Sea Level</b>	8
<b>Must Learn KQL Part 3: Workflow</b>	12
A Common KQL Workflow	14
<b>Must Learn KQL Part 4: Search for Fun and Profit</b>	18
<b>Must Learn KQL Part 5: Turn Search into Workflow</b>	22
<b>Must Learn KQL Part 6: Interface Intimacy</b>	26
Filtering through the table elements	28
Sorting results	28
Grouping results	29
Selecting columns to display	30
Select a time range	30
Charts	31
EXTRA	32
Save search queries	32
Share Queries!	33
Format query	35
Queries Galore	36
Exporting Queries	37
New Alert Rule	37
Pin to Dashboard or Workbook	37
Settings	39
In-UI Reference	40

Tabs.....	41
Keyboarding Shortcuts .....	42
Intellisense for the Win .....	42
Must Learn KQL Part 7: Schema Talk .....	44
Column Types .....	46
Back to the UI .....	47
Schema Area Focus.....	49
Must Learn KQL Part 8: The Where Operator .....	52
Hands-on Recommendations.....	52
Where Operator.....	53
Must Learn KQL Part 9: The Limit and Take Operators .....	57
Must Learn KQL Part 10: The Count Operator .....	60
Must Learn KQL Part 11: The Summarize Operator .....	64
Must Learn KQL Part 12: The Render Operator .....	68
Must Learn KQL Part 13: The Extend Operator.....	74
Must Learn KQL Part 14: The Project Operator .....	79
Must Learn KQL Part 15: The Distinct Operator .....	84
All YOU, baby! .....	86
Must Learn KQL Part 16: The Order/Sort and Top Operators .....	88
Must Learn KQL Part 17: The Let Statement .....	91
Creating Variables from Scratch.....	92
Creating Variables from Existing Data.....	93
Creating Variables from Microsoft Sentinel Watchlists .....	93
Must Learn KQL Part 18: The Union Operator.....	95
Must Learn KQL Part 19: The Join Operator .....	97
Must Learn KQL Part 20: Building Your First Microsoft Sentinel Analytics Rule .....	101
Analytics Rule.....	102

# Must Learn KQL Part 1: Tools and Resources

After hearing that our customers' largest barrier to using things like Defender, Microsoft Sentinel and even reporting for Intune is KQL, the query language, that was a wake-up call for me. And, of course, (if you know me) I want to do something about it. KQL is a beautifully simple query language to learn. And believe me – if I can learn it, there's no question that you can learn it. I feel bad that there's just not enough knowledge around it because I've taken for granted that everyone already had the proper resources to become proficient. But that's not the case.

Internally, plans are being developed now to make KQL learning a bigger focus and you'll see new education around this query language start to take shape in various areas on the Microsoft properties and elsewhere. So, that's good news for everyone.

There's bits and pieces already scattered about the Internet, but they are seemingly now difficult to identify and locate.

So, as a first step in a series that I'll be writing called “**Must Learn KQL**”, I want to supply some good resources that can be used to accomplish the other things I'll talk about going forward. Some of these I use every day. Some I use only when the need arises, but they're valuable, nonetheless. This is a working document, so expect updates over time. This is not a definitive list by any means, so if you have other resources not listed here that you find valuable and believe others would benefit, let me know and I'll add them in.

Stay tuned as I map out this series. Of course, since my area of forte at Microsoft is security, the series will be security focused. So, the knowledge you gain will help you with our security platforms but also anything data centric that utilizes KQL.

One last tidbit of a tip... I use Microsoft Edge's Collections feature quite a bit. This is an extremely useful tool for capturing and grouping topics. If you find any of the links below valuable, I suggest using Edge Collections so you can always come back to them later.

## Reference

[The code repository for this series \(GitHub\)](#)

[Kusto Query Language Reference Guide](#)

[Azure Monitor Logs table reference](#)

[Marcus Bakker's Kusto Query Language \(KQL\) – cheat sheet](#)

[SQL to Kusto cheat sheet](#)

[Splunk to Kusto Query Language map](#)

[Kusto Query Language in Microsoft Sentinel](#)

[Useful resources for working with Kusto Query Language in Microsoft Sentinel](#)

## Practice Environments

[Write your first query with Kusto Query Language \(Learn module\)](#)

[KQL Playground](#) – only need a valid Microsoft account to access.

[Data Explorer](#) – not security focused. Contains things like geographical data and weather patterns. Exercises for this can be found in the *Learn Azure Sentinel* book below.

## Actual Books

[Learn Azure Sentinel: Integrate Azure security with artificial intelligence to build secure cloud systems](#) – this book uses Data Explorer (see above) for hands-on exercises.

[Azure Sentinel in Action: Architect, design, implement, and operate Azure Sentinel as the core of your security solutions](#) – this book is the next edition of the one just above and also used Data Explorer for hands-on examples.

## Tools

[Kusto.Explorer](#) – a rich desktop application that enables you to explore your data using the Kusto Query Language in an easy-to-use user interface.

[Kusto CLI](#) – a command-line utility that is used to send requests to Kusto and display the results.

[Visual Studio Code](#) with the [Kusto extensions pack](#)

[Real-Time KQL](#) – eliminates the need to ingest data first before querying by processing event streams with KQL queries as events arrive, in real-time

[getschema operator](#) – As I noted in [Part 5](#) of this series: this is *the Rosetta stone of KQL operators*. When used, getschema displays the Column Name, Column Ordinal, Data Type, and Column Type for a table. This is important information for filtering data. [Part 5 talks about this](#).

## Blogs, Websites, and Social

[#MustLearnKQL](#) – the official Twitter hashtag of this series

[The #KQL hashtag on Twitter](#)

[The #365daysofkql hashtag on Twitter](#)

[Kusto King](#)

[The KQL Cafe](#) = podcast and community

[Matt Zorich's curated list of KQL learning resources](#)

## Video

[TeachJing's KQL Tutorial Series](#)

[Recon your Azure resources with Kusto Query Language \(KQL\)](#)

[How to start with KQL?](#)

[Azure Sentinel webinar: KQL part 1 of 3 – Learn the KQL you need for Azure Sentinel](#)

[Azure Sentinel webinar: KQL part 2 of 3 – KQL hands-on lab exercises](#)

[Azure Sentinel webinar: KQL part 3 of 3 – Optimizing Azure Sentinel KQL queries performance](#)

[Querying Azure Log Analytics \(with KQL\)](#)

## GitHub Query Examples

[My GitHub repo for Microsoft Sentinel KQL](#)

[The official Microsoft Sentinel repo](#)

[Wortell's KQL queries](#)

[Clive Watson's KQL queries and workbooks](#)

[Matt Zorich's \(the originator of the #365daysofkql Twitter hashtag\) KQL queries](#)

## Must Learn KQL Part 2: Just Above Sea Level

To start the journey learning KQL in this *Must Learn KQL* series, it's helpful to understand where the name KQL came from and why the reference makes so much sense. Once you understand the idea behind the query language, a lightbulb should go off and prepare you for the rest of the series through an expanded scope of learning capability.

Plus, not everyone knows about this, so you'll be the cool kid. And, if you ever play *Trivial Pursuit* and this question comes up, you'll win the pie piece and possibly the entire game. How can that not be good knowledge?

The question?

### Where does the name *Kusto* come from? (**K**usto **Q**uery **L**anguage)

To help explain this, I harken back to my childhood. Bear with me for a minute...

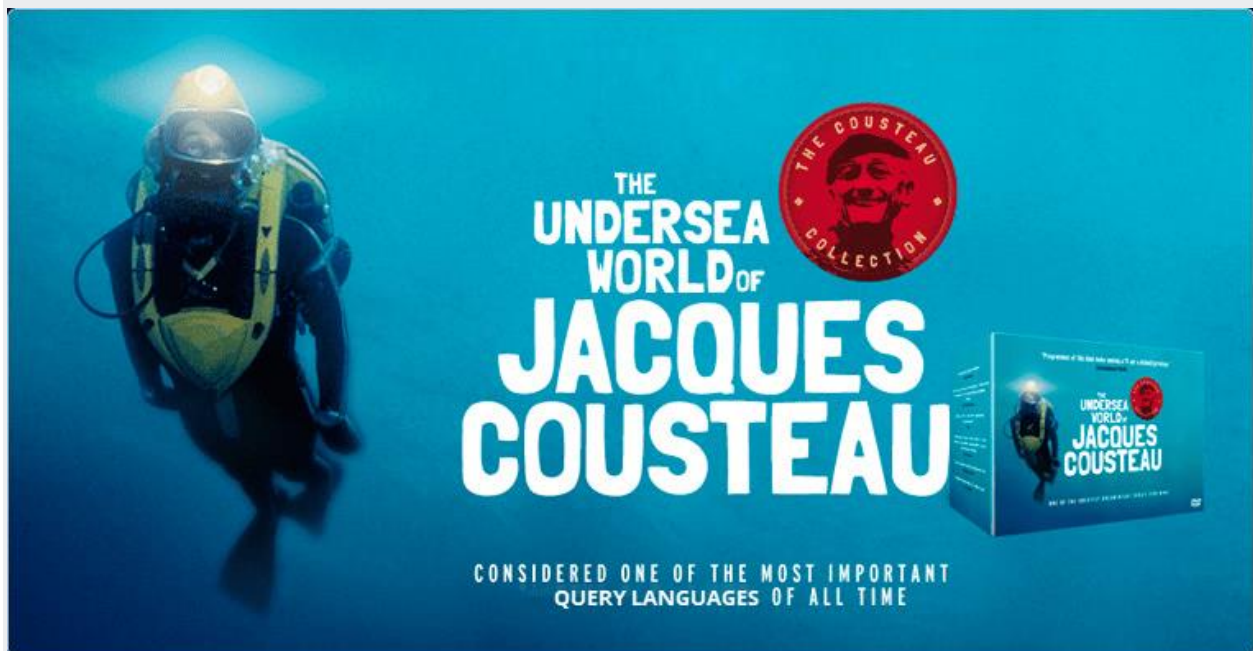
Growing up, my family was *one-of-those-families* that attended church anytime the church doors were open. As such, the majority of my parents' friends were at church. This meant that they would spend time before and after church services



catching up with their friends, sometimes in a local restaurant where they'd all gather to have pie and coffee. Of course, Facebook didn't exist then, so in-person connections were even more important. Well...and there was pie. My mom, in particular, wanted to catch up with everyone she hadn't seen in a few days so this meant that our round-trip from home to church and back could take 3-4 hours.

On Sunday nights this was particularly problematic for me in that I wanted to rush home to catch TV shows like the [Six Million Dollar Man](#), [The Magical World of Disney](#), [Mutual of Omaha's Wild Kingdom](#), and the TV show that's the topic of our discussion here: [The Undersea World of Jacques Cousteau](#)...

That's right. KQL is named after the undersea pioneer, Jacques Cousteau.

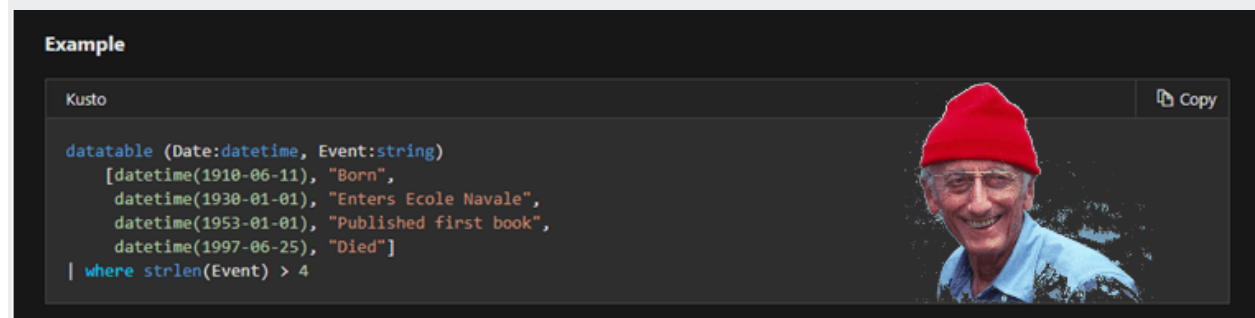


I loved this TV series. It was absolutely enthralling to me to understand that an entire world existed beneath the ocean waves and this unknown world was being brought to me by this wonderful, thick-accented explorer each week who dedicated his life to discovering what existed beneath the surface depths.

So, as you can imagine, I tried my dead-level best every Sunday night to rush my mom along. It didn't always work and was mostly just annoying, and you can bet I caught a few groundings from my insistence. But, still, this topic of discovering the undiscoverable drove me to concoct every type of machination imaginable to get home sooner on Sunday nights. I can't tell you the number of times I faked illness on Sunday afternoon in attempt to stay home Sunday night. And, as you can

imagine my mom quickly caught on and instituted a policy that if I stayed home on Sunday nights, I couldn't go to school on Monday. Which...at the time...I truly loved school, so that halted that plan. Give me a few years, and that wouldn't have worked. Timing is everything.

So, KQL is named after Jacques Cousteau. Even today, you can find evidence of this in our own Azure Monitor Docs. If you go to the [datatable operator page](#) right now, you'll still find a reference to Mr. Cousteau in an example that lists his date of birth, the date he entered the naval academy, when he published his first book entitled "[The Silent World: A Story of Undersea Discovery and Adventure](#)," and the date of his passing.



So, I hope you're catching on to this. If not, what is it that we are trying to accomplish when we query data tables for security purposes? What is that we're trying to accomplish though Hunting exercises and operations?

The answer? We are exploring the depths of our data. We are attempting to *surface* the critical and necessary security information that will tell us about potential exposure through simple, powerful queries.

Much like the story of the failed voyage of the Titanic. It wasn't the beautiful, pristine, easy-to-see and avoid iceberg mass that existed above the surface of the ocean that sunk the unsinkable ship and sent over 1,500 people to their grave. No, it was the huge mass under the surface that the captain and crew couldn't see and couldn't swerve to avoid that doomed the luxury passenger liner.



And, like that, it's the information that exists underneath the viewable rows and columns of data in our tables that we need to expose to identify threats and compromise and use to guard the gates. Just the initial rows and columns of exposed data isn't enough. We must delve into the depths of the data to find actionable information. And we need to do it quickly.

I hope all this makes sense.

It's as important to know *why* we do things, sometimes, as *how* to do them. Like Jacques Cousteau, security folks are explorers. We are mining the depths of the data no one sees to protect the environment against ever-growing and constantly evolving threats. We are discovering the undiscoverable.

KQL is an amazing and important piece of this capability. KQL was developed to take advantage of the power of the cloud through clustering and compute. Using this capability, KQL is designed as a well-performing tool to help surface critical data quickly. This is a big part of why it works so well and outshines many other query languages like it. KQL was built for the cloud and to be used against large data sets.

As a security person, you know that if a threat exists in the environment, you are on the clock to discover it, report it, investigate it, and remediate. A poorly performing query language can be the biggest barrier to that and become a security flaw. I've sat with customers who use other query languages and other SIEM-like tools that thought it was status quo that query results would take hours or sometimes days. When I showed that KQL produced those same results in seconds, they were astonished. So, the technology and infrastructure behind the query language is also critically important.

In the next post, I'll talk about the actual structure of a query. Even though the structure can deviate, understanding a common workflow of a KQL query can have powerful results and help you develop the logic needed to build your own workflows when it's time to create your own queries. In addition to being well-performing to enhance efficiency, the query language itself is simple to use and learn which, in turn, makes for more efficiency.

So, while we're *Just Above Sea Level* in this post (I hope you now appreciate the reference), we'll be using KQL as the sonar and diving bell to search the depths of our data.

## Must Learn KQL Part 3: Workflow

As I noted in [Part 2](#) of this *Must Learn KQL* series...

*Even though the structure can deviate, understanding a common workflow of a KQL query can have powerful results and help you develop the logic needed to build your own workflows when it's time to create your own queries.*

**Rod Trent, November 18, 2021**

The workflow (some folks call it *logic*, others call it *anatomy*, even others call it something else) is a big step into wrapping your mind around how to produce a KQL query. Just like a developer, assigning uniform, repeatable steps ensure you're not missing something and that your query results will produce the information you are looking to capture.

I tell customers all the time that it's not necessary to be a pro at creating KQL queries. It's OK not to be a pro on day 1 and still be able to use tools like Microsoft Sentinel to monitor security for the environment. If you understand the workflow of the query and can comprehend it line-by-line, you'll be fine. Because ultimately, the query is unimportant. Seriously. What's important for our efforts as security folks is the *results* of the query. The results contain the critical information we need to understand if a threat exists and then – if it does exist – how that threat occurred from compromise to impact.

Now, those that go on to develop their own queries and own Sentinel Analytics Rules after becoming a KQL pro will be much more capable. And that should be your goal, too. BUT don't get hung up on that. Again, it's about the results.

We've made it so crazily easy to share KQL queries that it's quite possible you may never have to create your own KQL query (*aside: I highly doubt it but COULD BE possible*).

In a future post in this series, I'll go over the actual interface you use to write and run the KQL queries in-depth but suffice to say that almost every service in Azure has a Logs *blade* (option in the Azure portal interface/menu) to accommodate querying that service's logs. This area provides for saving your queries, but also to *share* your queries.



```
1 SecurityEvent
2 | where EventID == "4688"
3 | project Computer, Account
4 | where Account == "WORKGROUP\\Windows365Sentinel\\..."
5 | summarize count() by Account
6
7
```

Account	count_
WORKGROUP\\Windows365Sentinel\\...	4,335

Share your queries

Because of this built-in capability, many of our customers regularly share their creations with each other, other colleagues, to their own blogs and GitHub repos, and even to the official Microsoft Sentinel GitHub repository (<https://aka.ms/ASGitHub>). In **Part 1** of this series, I supplied links to these and more. So, to prove my point...yes, it's absolutely possible you might not have to write your own KQL query for a long time.

So, because of that, it becomes even more critical that you at least understand the workflow. Again, if you can read a query line-by-line and determine that the results will produce what you are looking for, you're golden. If, through your newfound understanding, the query can't produce your requirements, you can modify it by line instead of a wholesale adaption. This should be your first KQL goal: read queries.

Through this series, I'll provide queries for you to use and get hands-on experience because I believe in learning by doing. We'll be using the links in the **Practice Environments** section in **Part 1** for the hands-on. But focus initially more on the structure and logical workflow.

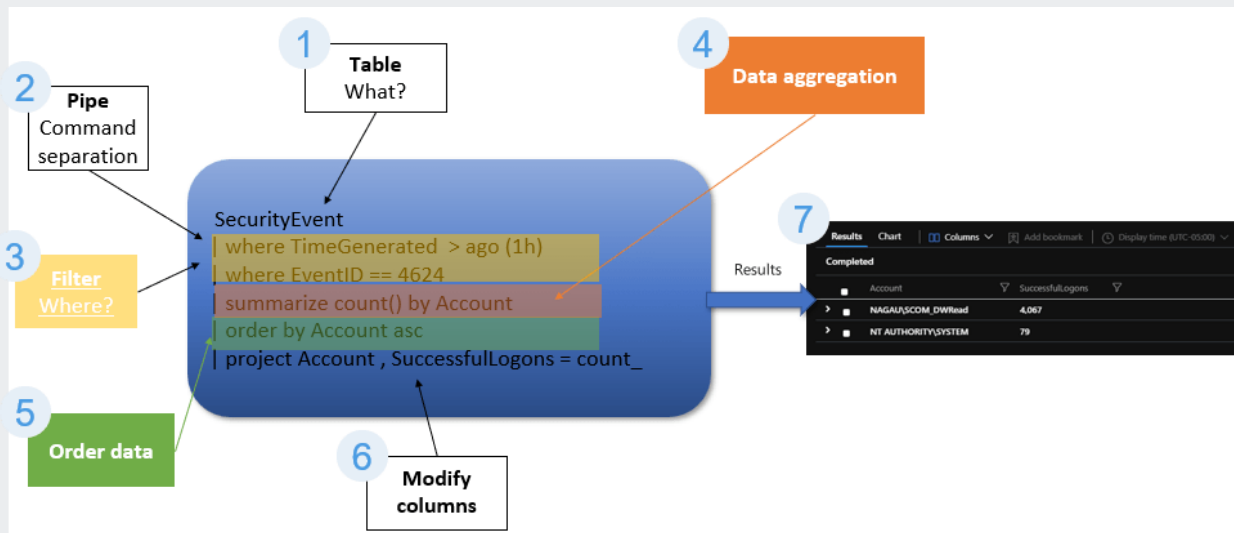
And, with that...

## A Common KQL Workflow

To get started on the journey to learning KQL, let's look at the standard workflow of a common search query. Not the *search operator* (I'll talk about in the next post), but the search query. This is the query structure we use to search, locate information, and produce results.

The following represents the common workflow of a KQL search query.

*P.S. I've enabled image linking in this post so you can click or tap to open the image in a larger view. So, you can open the image in a new window or new tab to better follow along.*



Let's break this query down by the steps.

1. The first step is to identify the table we want to query against. This table will contain the information that we're looking for. In our example here, we're querying the `SecurityEvent` table. The `SecurityEvent` table contains security events collected from windows machines by Microsoft Defender for Cloud or Microsoft Sentinel. For a full list of all services tables, see the [Azure Monitor Logs table reference](#) (also available in [Part 1](#)).
2. The **pipe** (`|`) character (the shifted key above the Enter key on most keyboards) is used to separate commands issued to the query engine. You can see here that each command is on its own line. It doesn't have to be this way. A KQL query can be all one single line. For our efforts, and as a recommendation, I prefer each command on its own line. For me, it's just neater and more organized which makes it easier to troubleshoot when a query fails or when I need to adjust the query to produce different results.

3. Next, we want to filter the data in some way. If I simply entered the table and ran that as its own, single query, it will run just fine. Doing that returns all rows and columns (up to a limit – which I believe is now 50,000 rows) of the data stored in the table. But our goal is getting exact data back. As an analyst looking for threats, we don't want to have to sift through 50,000 rows of data. No, we want to look for specific things. The **Where operator** is one of the best ways to accomplish this. You can see here in the example that I'm filtering first by when the occurrence happened (*TimeGenerated*) and then (remember the pipe character – *another line, another command*) by a common Windows Event ID (4624 – successful login).
4. The next step in our workflow is to provide data aggregation. What do we want to do with this filtered data? In our case in the example, we want to create a count of the Accounts (*usernames*) that produced a successful login (*EventID 4624*) in the last 24 hours (*TimeGenerated*).
5. Next let's tell the query engine how we want to order the results. Using the **Order operator**, I'm telling the query engine that when the results are displayed, I want it shown in alphabetical order by the Account column. The 'asc' in the query in the Order Data step is what produces this ordering. If we wanted descending order we'd use 'desc'. Don't worry, we'll dig deeper into each of these operators as we go along in the series.
6. Generally, the last thing that I'll do with this search query is tell the query engine exactly what data I want displayed. The **Project operator** is a powerful command. We'll dig deeper into this operator later in this series, but for our step here, I'm telling the query engine that after all my filtering, data aggregation, and ordering, I only want to display two columns in my results: Account and SuccessfulLogins

So, let's recap what this query accomplished...

It searched our stored security events in the *SecurityEvent* table for all Accounts that had a successful login in the last hour and chose to display only the Account and number of successful logins per Account in alphabetical order.

7. Our search query output is exactly that:



Results		Chart	Columns ▾	Add bookmark	Display time (UTC-05:00) ▾
Completed					
	Account	SuccessfulLogons			
>	NAGAU\SCOM_DWRead	4,067			
>	NT AUTHORITY\SYSTEM	79			

Search query output

See that? The Account column is in alphabetical order *ascending* and the SuccessfulLogons column shows how many times each Account successfully logged in.

If you need to, jump back through each step above until you get a good understanding of the workflow. Again, this is very common, and you'll see this structure many times working with Microsoft Sentinel and Defender products. Remember, it's about the results. If you can look at this example and get a good feel that you understand how the results were accomplished, line-by-line, you're on your way.

I invite you, though, to take this example and copy/paste it into a Logs environment to test. You can have this query to play with it in your own Microsoft Sentinel environment, or using the [KQL Playground](#) I provided as a resource in [Part 1](#).

```
SecurityEvent
| where TimeGenerated > ago (1h)
| where EventID == 4624
| summarize count() by Account
| order by Account asc
| project Account , SuccessfulLogons = count_
```

This query is also available from the GitHub repository for this blog series: <https://cda.ms/3fS>

I'd like to share one extra tidbit with you that you might find helpful as you start testing this KQL query example in your own, or our, environment.

Every language (scripting, coding, querying) has the capability to add comments or comment-out code through special characters. When the query, scripting, or development engine locates these characters, it just skips them. KQL has this same type of character. The character for KQL is the double forwardslash, or `//`

When you start testing this post's KQL query example, comment-out a line or two (put the double forwardslash at the beginning of the line) and rerun the query just to see how eliminating a single line can alter the results. You'll find that this is an important technique as you start developing your own KQL queries. I'll talk about this more later, too.

In the next post (Part 4) I'll talk through another, yet just as powerful, way to search for information using KQL that is a top pocket tool for Threat Hunters.

And, then I'll come back for Part 5 and show how to tie together both search methods to create the full operation of hunting to Analytics Rule. But don't worry, that's not the end. I have no clue how many parts this series be. A lot of it depends on you.

## Must Learn KQL Part 4: Search for Fun and Profit

Now that we have some understanding of the workflow (from [Part 3](#)) under our belts, I'm going to deviate from that for a brief minute in this post and then I'll bring it back together in Part 5 and combine Parts 4 and 5 to provide something extra meaningful to show you how it all fits together like an unsolved [Hardy Boys mystery novel](#). Hopefully, you're starting to see that my efforts here are logical and designed to accumulate enough knowledge that is necessary to move to the next plane of understanding.

What I want to do in this post, is give you something you can use today. When I'm done here, you should be able to take the knowledge and the query snippets to do your own hunting – or, rather, look inside your own environment to get an understanding of what is happening that's worth exposing and investigating.

One of the easiest ways to get started with KQL is the **search operator**. In [Part 3](#), I talked through the structure and workflow of a *search query*. In this post, I'll talk about the search *operator* (or command) and how it could be the most powerful KQL operator in the universe but will always be the best tool in the toolbelt to start any search operation.

Search is the first operator I reach for when trying to verify if something exists within the environment. In fact, our whole goal for using KQL as a security tool is to answer the following questions:

1. Does it exist?
2. Where does it exist?
3. Why does it exist?
4. *BONUS: There's a final question to this that's not part of this KQL series, but one that's important to the total equation and one that should be part of your SOC processes. That question is: How do we respond?*

If you click or tap the image to open it in a larger view, you'll see how the power of the search operator enables you to answer these questions.

It starts with an idea or theory that "something" exists in the environment. You may have gotten this idea from a dream or nightmare that someone in your organization is performing nefarious activities. But, most likely, the idea came from a news report or a post on social media from a trusted source about a nation-state actor being active with a new kind of ransomware.

Once these reports are available, someone (like Microsoft) will supply the Indicators of Compromise (IOCs) so you can search your environment to see if they exist. IOCs could be several things including filenames, file hashes, IP addresses, domain names, and more.

If they don't exist, you move on. If any of them do exist, you start to dig deeper to figure out where they exist, so you can, for example, quarantine systems or users, or block IP addresses or domains.

And, then you need to determine why they exist. Did a specific user click on something they shouldn't have clicked on in an email? Or did a threat actor successfully compromise a Domain Controller through control over a service or

elevate user account? Could it be that there is more impact on your environment than you originally thought?

All of this can be exposed through the simple process of search using the [search operator](#).

Let's walk through this together with a few simple queries that you can take and use to test your own environment. *(click or tap the image to open the larger version in a new browser tab to following along)*

The image displays three overlapping screenshots of the Microsoft Sentinel search interface, illustrating a search process:

- Step 1:** The search bar contains the query `search "rodtrent"`. The results table shows a single entry for `InsightsMetr...` with a `CPC-rodtrent E2` status.
- Step 2:** The search bar contains the query `search "rodtrent" | distinct Stable`. The results table shows a single entry for `Stable`.
- Step 3:** The search bar contains the query `search in (OfficeActivity) "rodtrent"`. The results table shows multiple entries for `OfficeActivity` with various `ExchangeItem` and `MailItemsAccessed` operations.

Who, What, When, Where?

In *step 1* in the image, I'm performing a simple search for a username. In this case, it's an ego search – I'm searching in my own environment for my own activity. This could be an IOC that you want to search for. Just replace my name with the string of text you want to expose in the results.

```
search "rodtrent"
```

As you can see in the image, my search produced results, telling me that this *thing* I searched for *does* exist in my environment.

Since it does exist, I want to understand where it exists. I do this by making a simple adjustment to my original query by adding a line that tells the query engine to just show me the specific tables that my IOC exists in. This will give me a good indication of what type of activity it was. *Step 2* shows...

search "rodtrent"

| distinct \$table

Let’s assume that I’m looking for user activity because the reported threat is malware. I know that user activity is most generally recorded and contained in a few places including Microsoft Office and Defender for Endpoint.

In my example in *Step 3* in the image, I’ve adjusted my search operator query to focus only on the OfficeActivity table. Here what that looks like:

search in (OfficeActivity) "rodtrent"

Now that I have my results of rodtrent’s activity in the OfficeActivity table, I can begin sifting through the rows and columns of data to learn more about the occurrence and to start to tune my query even more.

	TimeGenerated [Local Time]	\$table	RecordType	Operation	OrganizationId	OrganizationId_
>	11/22/2021, 7:27:07.000 AM	OfficeActivity	ExchangeItem	MailItemsAccessed	f70d46d0-7fd7-48a5-8586-e6a8199d...	f70d46d0-7fd7-48a5-
>	11/22/2021, 7:27:21.000 AM	OfficeActivity	ExchangeItem	MailItemsAccessed	f70d46d0-7fd7-48a5-8586-e6a8199d...	f70d46d0-7fd7-48a5-
>	11/22/2021, 7:27:20.000 AM	OfficeActivity	ExchangeItem	MailItemsAccessed	f70d46d0-7fd7-48a5-8586-e6a8199d...	f70d46d0-7fd7-48a5-
▼	11/22/2021, 7:27:21.000 AM	OfficeActivity	ExchangeItem	MailItemsAccessed	f70d46d0-7fd7-48a5-8586-e6a8199d...	f70d46d0-7fd7-48a5-
...						
	\$table	OfficeActivity				
	TenantId	e73fcae6-0260-4da5-9d56-f9e36d6db671				
	RecordType	ExchangeItem				
	TimeGenerated [UTC]	2021-11-22T12:27:21Z				
	Operation	MailItemsAccessed				
	OrganizationId	f70d46d0-7fd7-48a5-8586-e6a8199d4de5				
	OrganizationId_	f70d46d0-7fd7-48a5-8586-e6a8199d4de5				
	UserType	Regular				

Results from the OfficeActivity table

When we come back for [Part 5](#), I'll show you how to turn your search query into a workflow like I talked about in [Part 3](#).

One last thing for this post. I mentioned that user activity is generally reported from the Microsoft Office and Defender for Endpoint tables. I've given you examples for searching the OfficeActivity table. But Defender for Endpoint is more than one table. In fact, Defender for Endpoint consists of the following 10 tables: DeviceEvents, DeviceFileCertificateInfo, DeviceFileEvents, DeviceImageLoadEvents, DeviceInfo, DeviceLogonEvents, DeviceNetworkEvents, DeviceNetworkInfo, DeviceProcessEvents, and DeviceRegistryEvents.

Fortunately, the KQL [search operator](#) supports the wildcard character. So, you can search for those IOCs across the entire Defender for Endpoint solution by doing the following:

```
search in (Device*) "rodtrent"
```

And, incidentally, if you have the Defender for 365 Data Connector enabled for Microsoft Sentinel and you enable the Microsoft Defender for Office 365 logs, the OfficeActivity table isn't the only Microsoft Office data you can query. Enabling these logs gives you access to EmailEvents, EmailUrlInfo, EmailAttachmentInfo, and EmailPostDeliveryEvents tables which means you can take advantage of the search operator's wildcard capability here, too.

*All the query code in this post is contained in the series' GitHub repo here: <https://cda.ms/3gG>*

P.S. Enjoying this series? Share it with someone!

## Must Learn KQL Part 5: Turn Search into Workflow

Now, that we've talked about using the [Search operator in Part 4](#) to answer those three basic SOC analyst questions of: 1) Does it exist? 2) Where does it exist? and, 3) Why does it exist?, we can take that learning and the results of that type of query and meld it with the standard search query structure I talked about in [Part 3](#).

In [part 4](#), I ended with a query to locate activity by a user called "rodtrent". I found that this *rodtrent* person had performed potentially strange activity in the OfficeActivity table (the table for Office 365 activity) that needs to be checked out. As shown, the [search operator](#) is a powerful tool to find things of interest. The

results of the search operator query were thousands of rows of data. That's inefficient.

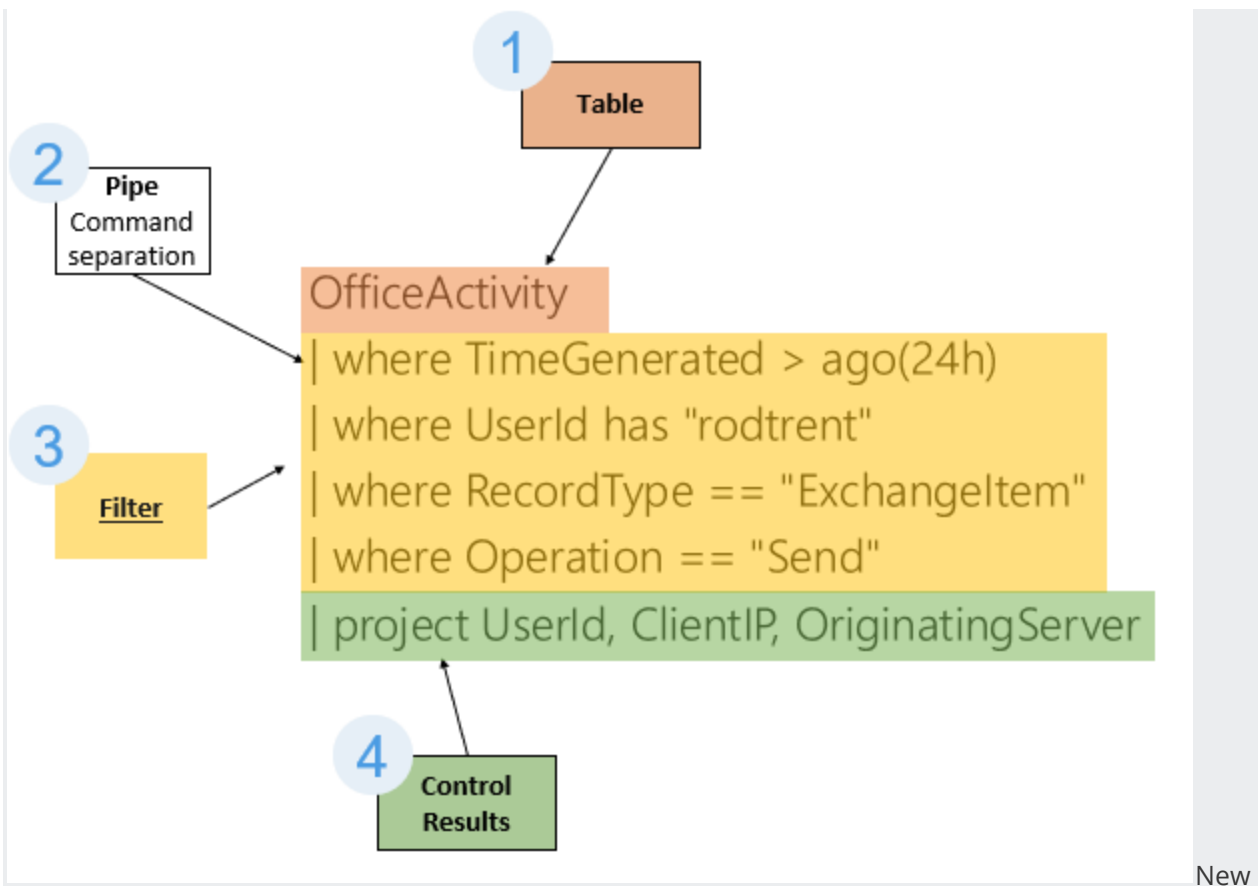
So, now that we've found something interesting, we want to use the structure of the Search Query to pare down the results to minimize the effort and workload to identify that that *something interesting* is something notable and worth investigating.

If you need to, open up [Part 3](#) in a new Window or browser Tab to review the Search Query Workflow as I walk through the next section.

In the following example, note that this is a non-issue situation, but I want to start with a basic Search query before we start building toward more complex queries in future posts to get a fully rounded understanding of the “why” behind why we do this. The one below is even simpler than the one discussed in [Part 3](#) where I also talk about aggregating and ordering data. I'll come back to those concepts later, particularly when I get into creating your own in-query visualizations like pie and bar charts. No, for our efforts in this post, I want to focus on how easy it is to filter the data. Again, KQL isn't hard, and some of your most powerful queries may only be a few lines of code.

Turning your hunting operations into more formal Search structure queries is the building blocks for creating your own Analytics Rules in Microsoft Sentinel. Analytics Rules should be precise logic to enable your operations to focus exactly where it needs to focus; and because, capturing data outside of what was intended is both inefficient and problematic for isolating actual security events.

The example (*available from the series' GitHub repo at: <https://cda.ms/3jd>*):



search query

New

Let's break this new Search query down together like was done in [Part 3](#). This one, again, is even a tad bit simpler than when describing the Search workflow, but as you'll see, it's the [where operator](#) that is sometimes our biggest, most powerful, and best workhorse and pal for tuning efficient results.

1. The first step in our workflow is to query the OfficeActivity table. If you remember, from our time together in [Part 4](#), we're looking for user activity (in our case the user "rodtrent") in Microsoft Office.
2. As per the discussion in [Part 3 on workflow](#), I want to highlight the importance of the pipe command once again. I don't rehash the importance here. If you missed it, jump to [Part 3](#) to catch up.
3. In step 3 of the new Search query, I'm filtering how the query engine searches. I'm first telling to only look at data in the last 24 hours (TimeGenerated), then only looking through a column called UserId for the string "rodtrent", then telling the query engine to only capture Exchange activity from the RecordType data column, and finally



pinpointing the search to only Send operations. So, essentially, I'm looking for any emails that rodrent sent in the last 24 hours.

- Filtering the data is the key to everything. <= Read that again. Filtering the data that is returned produces exact, actionable data. It also improves the results performance of our queries. Where the search operator may return thousands of rows of data in 15 seconds (or less), by properly filtering the data to return exactly what is necessary returns just the number of rows of data we asked for which greatly improves the processing time. Where the search operator may have taken 15 seconds, our new Search structure query will take 5 seconds or less. The Where operator is the key to this operation. Learn it. Know it. Keep the Where operator reference page handy: <https://cda.ms/3jh>.

4. Finally, I'm using the [project operator](#) to control exactly what is show in the results window. In this case, I only want to show the user, the user's IP address, and the server where the email originated from.

The results?

The screenshot shows a search query interface with a dark theme. At the top, there's a toolbar with buttons for 'Run', 'Time range: Set in query', 'Save', 'Share', 'New alert rule', 'Export', and 'Pin to dashboard'. Below the toolbar, the query is written in a monospace font:

```
1 OfficeActivity
2 | where TimeGenerated > ago(24h)
3 | where UserId has "rodrent"
4 | where RecordType == "ExchangeItem"
5 | where Operation == "Send"
6 | project UserId, ClientIP, OriginatingServer
7
```

Below the query, there's a section for 'Results' with tabs for 'Results' and 'Chart'. The 'Results' tab is active, showing a table of results. The table has columns for 'UserId', 'ClientIP', and 'OriginatingServer'. There are two rows of results, both for the user 'rodrent@sixmilliondollarman.onmicrosoft.com'.

UserId	ClientIP	OriginatingServer
rodrent@sixmilliondollarman.onmicrosoft.com	40.114.40.132	CH2PR04MB7000 (15.20.4200.000)
rodrent@sixmilliondollarman.onmicrosoft.com	40.114.40.132	CH2PR04MB7000 (15.20.4200.000)

Search query results

As you can plainly see in the query results, this matches exactly what my query proposed.

**EXTRA:** We saw in [Part 4](#) with our Search operator, how results from our queries are in named rows and columns of data. And, you see here in this post, how I'm constantly filtering against known column names in the tables. Some might wonder how I come up with those schema names. Of course, it helps that I work with these tables constantly, but I do have a couple secrets to share. First off, as noted in [Part 1](#), I use the [Azure Monitor Logs table reference](#) quite a bit. However, there's also the Rosetta stone of KQL operators: [getschema](#)

Running a simple...

OfficeActivity
getschema

...will produce a list of all the named columns of a specific table. The example above displays all the named columns of the OfficeActivity table. Each of these columns can be used in your [where operator](#) filtering efforts.

...

In this post, I've given you a simple query to practice with. In [Part 6](#), I'll come back and dig into the actual interface for developing your own queries (instead of just running the ones I've given).

## Must Learn KQL Part 6: Interface Intimacy

I preface this post by saying this: everything discussed in this post about the User Interface (UI) can be done (and *should* be done, eventually) in the KQL query itself.

When you're just starting with KQL, the UI can be a blessing. As you get further in your learning and comfortability with the query language, it can be a crutch – particularly when you need to find something quickly because of a perceived security threat and view it in a way that's most meaningful. Still, understanding the UI's capabilities is important.

In this post, I'll give you a whirlwind tour of the UI, but again with the assumption that, eventually, every action it provides I'll cover on how to accomplish it using KQL as we get further and further along in this series.

The Logs blade exists in almost every Azure service, allowing you to query the activity logs for that service. For our purposes for Microsoft Sentinel, since all of those services' (and more) logs are consolidated in the Log Analytics workspace for Microsoft Sentinel, we get to use the UI to query everything. It can be a bit of a power rush.

For those that already have deep-level experience with the Logs UI in Azure services, this may not be your favorite part of this series, but you also may learn something you missed or that's been updated recently, so make sure not to overlook anything important. And, please, please, **PLEASE** – if you're the expert in the UI and with KQL, pass this along to someone who needs it.

Like everything in Azure, there's updates and enhancements constantly, so I'll try to keep this part of the series up-to-date continually. My youngest son is the epitome of FOMO (fear of missing out) and I feel like him sometimes when I've been away from the Azure portal or the Microsoft Sentinel console for even a day. Every day can be a new adventure. As a customer, you might think, or even become frustrated that it's hard to keep up with all the changes going on in the Azure services and other products. But, believe me, those of us that work at Microsoft are faced with the exact same scenario and the same difficulties in keeping up-to-date. So, we can help each other in this respect. See something in this part of the series that's slightly off or maybe improved? Or, maybe I've chosen not to cover an area or feature that you need more knowledge about. Let me know and I'll get it updated toot sweet.



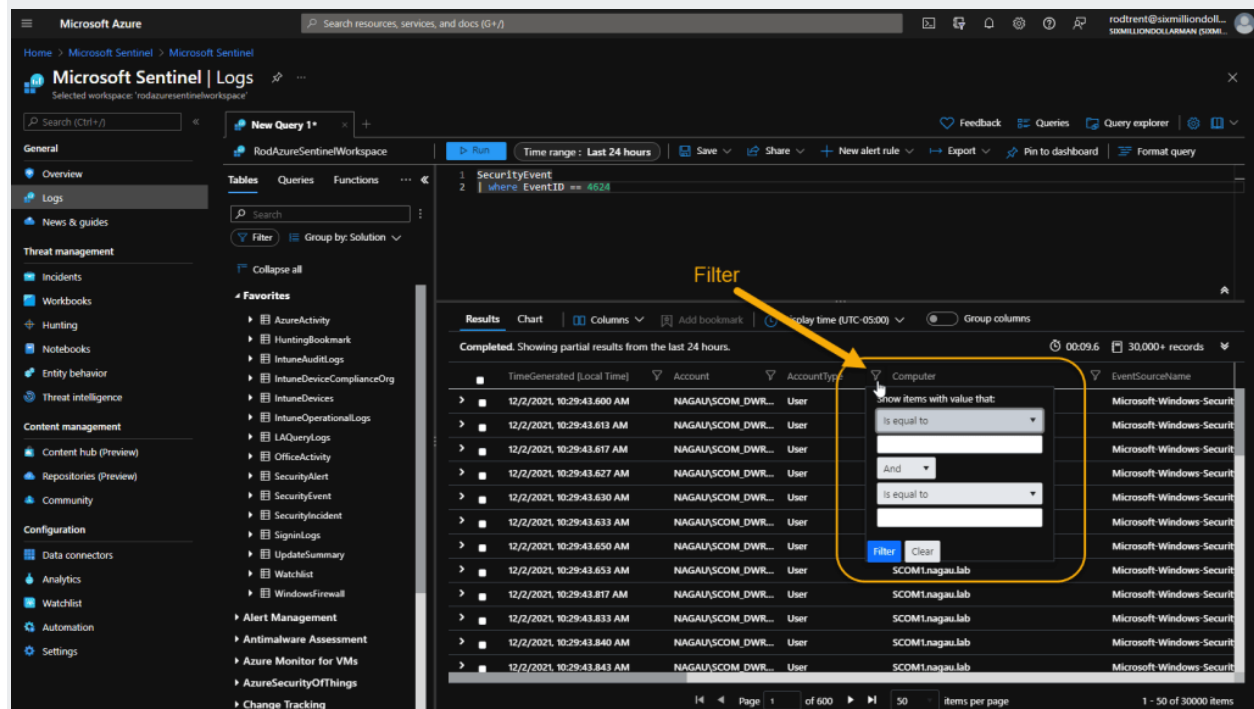
**HANDS-ON:** If you'd like to follow along yourself with the UI areas and descriptions in this post (instead of just reading through them in the text), use the [KQL Playground](#) that is referenced as a **Practice Environment** in the resources list of [Part 1](#).

I'll start this part of the series talking about those areas in the UI that are most important to our efforts in learning how to manipulate the KQL query data, and then follow up with the rest of the interface in the **Extras** section below, so you get the full intimate affair. And don't forget to come back for Part 7 for the **Schema Talk** (see the [TOC](#)) where I'll finish up covering the UI with those areas of the UI that pertain to working with the tables.

OK...so let's dig in...

## Filtering through the table elements

To focus on a specific column, select the Filter icon, then select values to adjust the results display.



The screenshot shows the Microsoft Sentinel Logs interface. A KQL query is entered in the query bar: `SecurityEvent | where EventID == 4624`. The results table displays columns: TimeGenerated (Local Time), Account, AccountType, Computer, and EventSourceName. A filter is applied to the 'Computer' column, showing a dropdown menu with the text 'Show items with value that:' and a selection of 'is equal to'. The results table shows 30,000+ records, with the first 10 items displayed. The 'Computer' column values are SCOM1.nagau.lab.

The example query in the above and following images is located here: <https://cda.ms/3mD>

## Sorting results

To sort the results by a specific column, such as timestamp, click the column title. One click sorts in ascending order while a second click will sort in descending. An arrow will display in the column next to the column title to show which direction the results are sorted.

Microsoft Azure

Search resources, services, and docs (G+)

Home > Microsoft Sentinel > Microsoft Sentinel

Microsoft Sentinel | Logs

Selected workspace: 'rodazuresentinelworkspace'

Search (Ctrl+F)

New Query 1\*

RodAzureSentinelWorkspace

Time range: Last 24 hours

1 SecurityEvent  
2 where EventID == 4624

Sort by ascending and descending order

Results Chart Columns Add bookmark Display time (UTC-05:00) Group columns

Completed. Showing partial results from the last 24 hours.

TimeGenerated [Local Time]	Account	AccountType	Computer	EventSourceName
12/2/2021, 10:58:06.157 AM	NAGAU\SCOM_DWR...	User	SCOM1.nagau.lab	Microsoft-Windows-Secur...
12/2/2021, 10:58:05.097 AM	NAGAU\SCOM_DWR...	User	SCOM1.nagau.lab	Microsoft-Windows-Secur...
12/2/2021, 10:58:05.093 AM	NAGAU\SCOM_DWR...	User	SCOM1.nagau.lab	Microsoft-Windows-Secur...
12/2/2021, 10:58:05.053 AM	NAGAU\SCOM_DWR...	User	SCOM1.nagau.lab	Microsoft-Windows-Secur...
12/2/2021, 10:58:05.050 AM	NAGAU\SCOM_DWR...	User	SCOM1.nagau.lab	Microsoft-Windows-Secur...
12/2/2021, 10:58:05.047 AM	NAGAU\SCOM_DWR...	User	SCOM1.nagau.lab	Microsoft-Windows-Secur...
12/2/2021, 10:58:05.043 AM	NAGAU\SCOM_DWR...	User	SCOM1.nagau.lab	Microsoft-Windows-Secur...
12/2/2021, 10:58:05.027 AM	NAGAU\SCOM_DWR...	User	SCOM1.nagau.lab	Microsoft-Windows-Secur...
12/2/2021, 10:58:05.023 AM	NAGAU\SCOM_DWR...	User	SCOM1.nagau.lab	Microsoft-Windows-Secur...
12/2/2021, 10:58:05.017 AM	NAGAU\SCOM_DWR...	User	SCOM1.nagau.lab	Microsoft-Windows-Secur...
12/2/2021, 10:58:04.983 AM	NAGAU\SCOM_DWR...	User	SCOM1.nagau.lab	Microsoft-Windows-Secur...
12/2/2021, 10:58:04.950 AM	NAGAU\SCOM_DWR...	User	SCOM1.nagau.lab	Microsoft-Windows-Secur...

Page 1 of 600 50 items per page 1 - 50 of 30000 items

## Grouping results

To group the results, first toggle the *Group Columns* option, then simply click and hold and drag the column header above the other columns.

Microsoft Azure

Search resources, services, and docs (G+)

Home > Microsoft Sentinel > Microsoft Sentinel

Microsoft Sentinel | Logs

Selected workspace: 'rodazuresentinelworkspace'

Search (Ctrl+F)

New Query 1\*

RodAzureSentinelWorkspace

Time range: Last 24 hours

1 SecurityEvent  
2 where EventID == 4624

Group columns

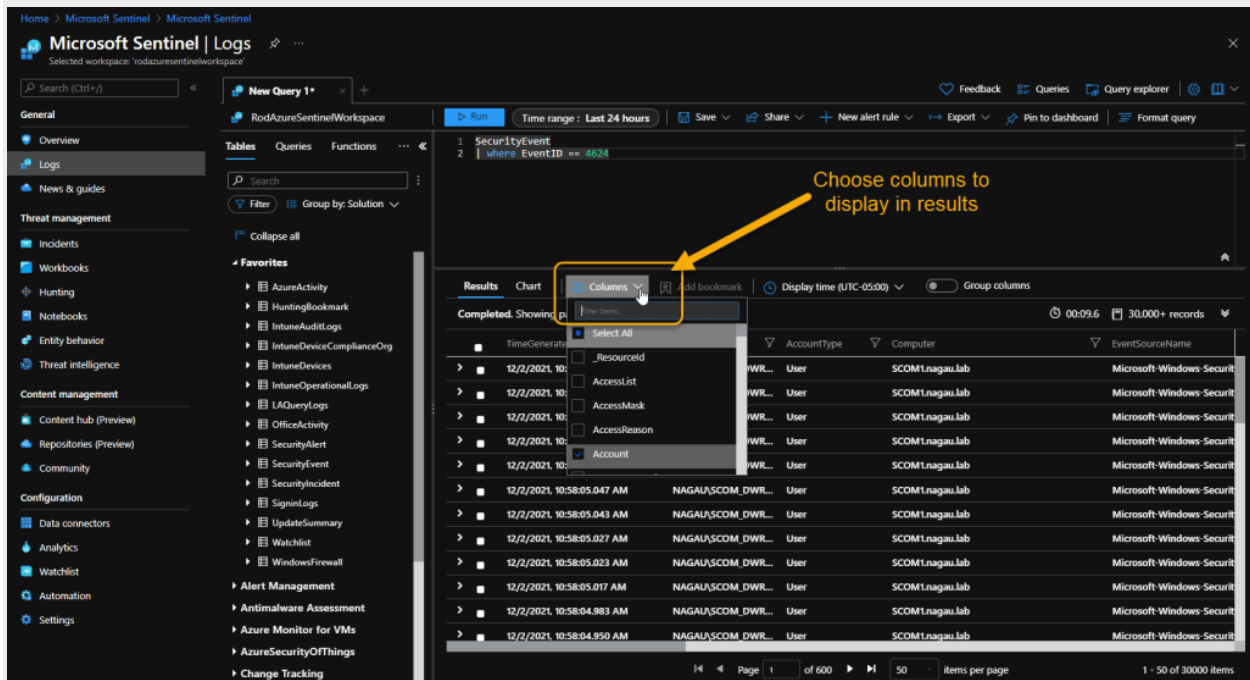
Completed. Showing 3 results from the last 24 hours.

AccountType	TimeGenerated [Local Time]	Account	AccountType	Computer	EventSourceName
Machine	12/2/2021, 10:57:20.790 AM	NT AUTHORITY\SYSTEM	Machine	SCOM1.nagau.lab	Microsoft-Windows-S...
Machine	12/2/2021, 10:57:15.063 AM	NT AUTHORITY\SYSTEM	Machine	CPC-rodrent-E2	Microsoft-Windows-S...
Machine	12/2/2021, 10:57:07.047 AM	NT AUTHORITY\SYSTEM	Machine	SCOM1.nagau.lab	Microsoft-Windows-S...

Page 1 of 600 50 items per page 1 - 50 of 30000 items

## Selecting columns to display

To add and remove a column that is displayed select the Columns button.



The screenshot shows the Microsoft Sentinel Logs interface. On the left is a navigation pane with categories like General, Overview, Logs, Threat management, and Configuration. The main area displays a query for 'SecurityEvent' with the filter 'where EventID == 4634'. Below the query, the 'Columns' button is highlighted with a yellow box and an arrow pointing to it with the text 'Choose columns to display in results'. The 'Columns' dropdown menu is open, showing a list of columns with checkboxes: 'TimeGenerated', 'AccountType', 'Computer', 'EventSourceName', 'Account', 'AccessList', 'AccessMask', 'AccessReason', and 'ResourceId'. The 'Account' checkbox is checked. The table below shows columns for 'TimeGenerated', 'AccountType', 'Computer', and 'EventSourceName'. The table has 30,000+ records.

TimeGenerated	AccountType	Computer	EventSourceName
12/2/2021 10:58:05.047 AM	User	SCOM1.nagau.lab	Microsoft-Windows-Security...
12/2/2021 10:58:05.043 AM	User	SCOM1.nagau.lab	Microsoft-Windows-Security...
12/2/2021 10:58:05.027 AM	User	SCOM1.nagau.lab	Microsoft-Windows-Security...
12/2/2021 10:58:05.023 AM	User	SCOM1.nagau.lab	Microsoft-Windows-Security...
12/2/2021 10:58:04.983 AM	User	SCOM1.nagau.lab	Microsoft-Windows-Security...
12/2/2021 10:58:04.950 AM	User	SCOM1.nagau.lab	Microsoft-Windows-Security...

You will notice when you work with this UI feature, there's a number of columns that are omitted from the results display. There's some intelligence built in that looks at the table data and only shows results that it deems pertinent to the operation – in our case, that operation is security monitoring. It also locates columns that contain no data and omits these from the display. All these measures are intended functions to help build efficiency and eliminate unnecessary data, but also to improve query results performance. But, using this feature (and actual KQL operators like *project* we'll talk about later), you can use the UI to pick and choose what to review.

## Select a time range

To add a custom time range, select the *Time range* option.

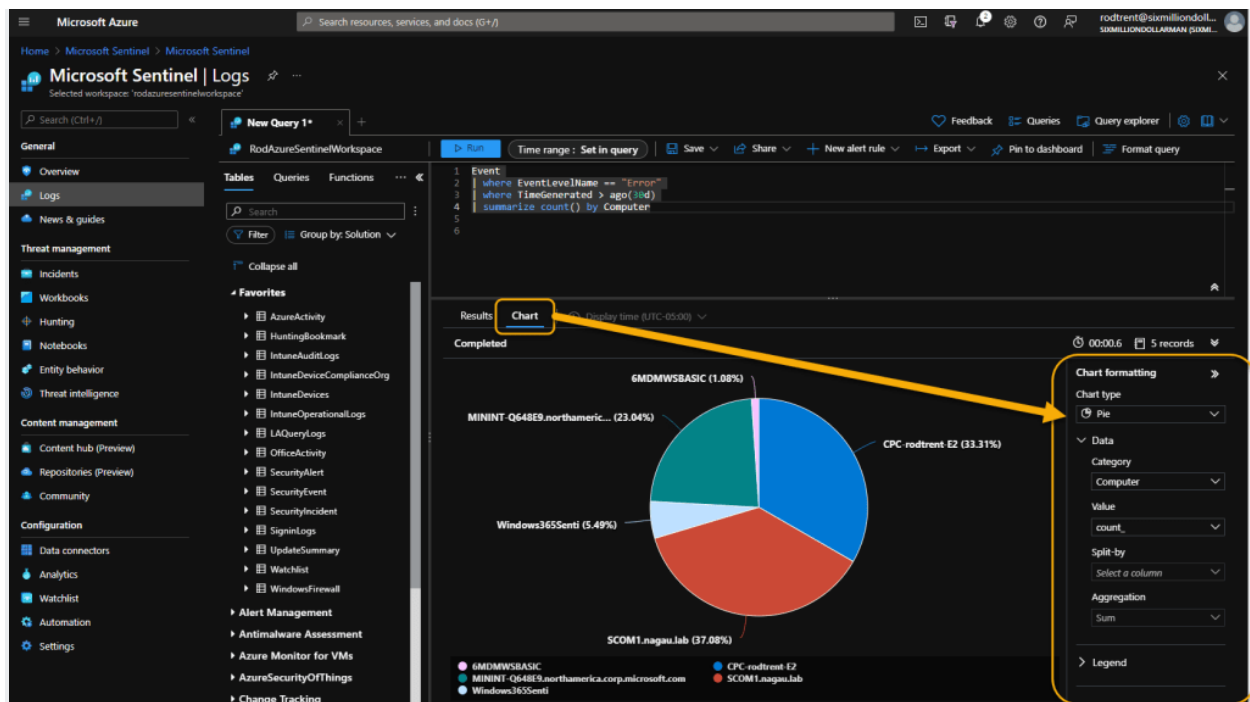


Incidentally, 24 hours is the default for Microsoft Sentinel. Each time you enter the console or attempt to work with KQL in the Logs blade, it will default to this time value. This is based on security principles that a SOC or security teams should be focused on the most current data. Responsibilities, tasks, policies, and procedures of a well-tuned security team should ensure that all current events are monitored and managed in some way at the end of each day so that they are ready for the next round of events. That's not always the case, of course, but that's one reason why Microsoft Sentinel always defaults to 24 hours.

## Charts

To add a chart as a visual format you can select the CHART option just about the results window at the bottom of the UI. On the right-hand side you have many options for manipulating the visual aspect the data.





The example query in the image above is available from here: <https://cda.ms/3mF>

Note that charting is dependent on tabular data. I'll talk about this when we get to the summarize, render, and bin operators in this series. (See the [TOC](#))

## EXTRA

In the previous section, I've discussed those areas in the UI that are going to help you manipulate the results. Again, while those are important areas, I'll show how to accomplish each of those using actual KQL query operators, so you don't have to rely on the UI.

You might notice I didn't spend any time talking about the *Tables*, *Queries*, and *Functions* areas in the Logs blade. I'll actually come back to those in Part 7 when I talk about the schema. (See the [TOC](#))

But before closing out this part of the series, I do want to also highlight some other cool areas of the UI that you might enjoy and have fun with.

### Save search queries

You can save your queries to *Query Packs* and then look them up and use them later. For more information on Query Packs, see: [Query packs in Azure Monitor Logs](#) and [How to Save an Azure Sentinel Query to a Custom Query Pack](#)



The screenshot shows the Microsoft Sentinel Logs interface. A query is being executed, and the results are displayed in a table. A yellow box highlights the 'Save' button in the top right corner of the query editor. An arrow points from this button to a 'Save as query' dialog box that is open on the right side of the screen. The dialog box has fields for 'Query name' and 'Description', and a checkbox for 'Save as Legacy query'. The 'Path' section is set to 'Save to the default query pack'. The 'Tags' section shows 'Resource type' as 'Log Analytics workspaces' and 'Category' as '0 selected'. The 'Label' section shows '0 selected' and a 'Create new label' link. The 'Save' button is at the bottom of the dialog box.

TimeGenerated (Local Time)	Account	AccountType	Computer
12/2/2021, 11:08:54.037 AM	NAGAU\$COM_DWRead	User	SCOM1.nagau.lab
12/2/2021, 11:08:54.070 AM	NAGAU\$COM_DWRead	User	SCOM1.nagau.lab
12/2/2021, 11:08:54.077 AM	NAGAU\$COM_DWRead	User	SCOM1.nagau.lab
12/2/2021, 11:08:54.080 AM	NAGAU\$COM_DWRead	User	SCOM1.nagau.lab
12/2/2021, 11:08:54.087 AM	NAGAU\$COM_DWRead	User	SCOM1.nagau.lab
12/2/2021, 11:08:54.107 AM	NAGAU\$COM_DWRead	User	SCOM1.nagau.lab
12/2/2021, 11:08:54.130 AM	NAGAU\$COM_DWRead	User	SCOM1.nagau.lab
12/2/2021, 11:09:17.727 AM	NT AUTHORITY\SYSTEM	Machine	SCOM1.nagau.lab
12/2/2021, 11:09:21.457 AM	NT AUTHORITY\SYSTEM	Machine	SCOM1.nagau.lab
12/2/2021, 11:11:24.377 AM	NAGAU\$COM_DWRead	User	SCOM1.nagau.lab
12/2/2021, 11:11:24.383 AM	NAGAU\$COM_DWRead	User	SCOM1.nagau.lab
12/2/2021, 11:11:24.390 AM	NAGAU\$COM_DWRead	User	SCOM1.nagau.lab

## Share Queries!

Sharing your fabulous query creations is an important capability for a number of reasons and not just for an ego boost or pat on the back when bragging to friends and colleagues.

The screenshot shows the Microsoft Sentinel Logs interface. A query is being executed, and the results are displayed in a table. A yellow box highlights the 'Share' button in the top right corner of the query editor. A dropdown menu is open, showing options: 'Copy link to query', 'Copy query text', 'Copy results', and 'Share to community'. The 'Share to community' option is highlighted. The results table shows the same data as the first screenshot.

TimeGenerated (Local Time)	Account	AccountType	Computer
12/2/2021, 11:08:54.037 AM	NAGAU\$COM_DWRead	User	SCOM1.nagau.lab
12/2/2021, 11:08:54.070 AM	NAGAU\$COM_DWRead	User	SCOM1.nagau.lab
12/2/2021, 11:08:54.077 AM	NAGAU\$COM_DWRead	User	SCOM1.nagau.lab
12/2/2021, 11:08:54.080 AM	NAGAU\$COM_DWRead	User	SCOM1.nagau.lab
12/2/2021, 11:08:54.087 AM	NAGAU\$COM_DWRead	User	SCOM1.nagau.lab
12/2/2021, 11:08:54.107 AM	NAGAU\$COM_DWRead	User	SCOM1.nagau.lab
12/2/2021, 11:08:54.130 AM	NAGAU\$COM_DWRead	User	SCOM1.nagau.lab
12/2/2021, 11:09:17.727 AM	NT AUTHORITY\SYSTEM	Machine	SCOM1.nagau.lab
12/2/2021, 11:09:21.457 AM	NT AUTHORITY\SYSTEM	Machine	SCOM1.nagau.lab
12/2/2021, 11:11:24.377 AM	NAGAU\$COM_DWRead	User	SCOM1.nagau.lab
12/2/2021, 11:11:24.383 AM	NAGAU\$COM_DWRead	User	SCOM1.nagau.lab
12/2/2021, 11:11:24.390 AM	NAGAU\$COM_DWRead	User	SCOM1.nagau.lab

There are four sharing options:

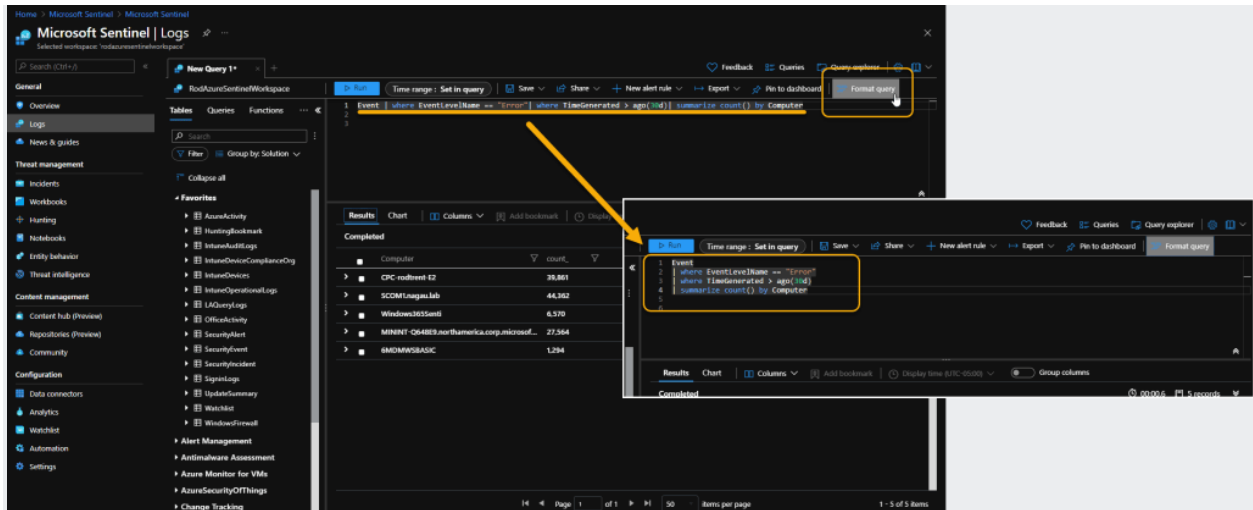
1. **Copy link to query:** Since the Azure portal and Microsoft Sentinel console is web-based, you can share the direct URL to the query you created by pasting it somewhere (email, Teams chat or channel, etc.). When you share the link and someone with proper access clicks on it, they are taken directly to the Logs blade and the query is run, so they can review the same results. This is an awesome team activity where you can get an extra set of eyeballs on a potential situation.
2. **Copy query text:** This function just copies the query itself so you can send that somewhere (to a team member, to a GitHub repo, etc.)
3. **Copy results:** Right now, this function literally does the exact same thing as the *Copy link to query* option. So, we'll put a pin here for when this changes in the future.
4. **Share to community:** This option is super-fantastic! By utilizing this sharing feature, the query you've created is copied and placed into an email template that is addressed to the Azure Monitor team at Microsoft. By submitting this after entering the requested information in the email template fields, your creation will be vetted and published to the GitHub repository for the Azure Monitor community! Imagine your name in lights, idolized for your contributions that helped solve the latest global security threat!



And, by the way, you can also submit your KQL creations to the official GitHub repository for Microsoft Sentinel. See **Add in your new or updated contributions to GitHub** for steps on how to accomplish that.

## Format query

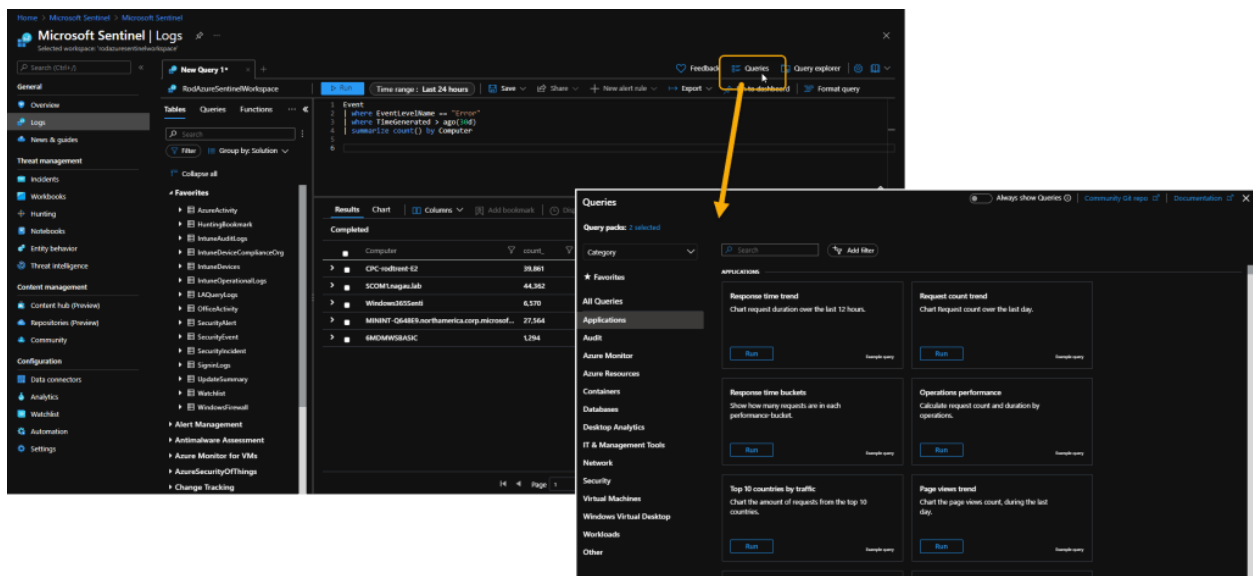
A super-cool, super-useful tool is the Format button in the UI. This button takes a badly formatted query and reformats it so it: a) works, or b) is in a more uniform, more readable format.



As I noted in [Part 3](#) about Workflow, because of the power of the pipe (|) command separator, a KQL query can be a single line of code. But that's a bit useless if you want to be able to determine what the query's intent is or need to debug it. This option turns it into a better format.

## Queries Galore

In addition to all the awesome KQL query goodness available from all over the Internet, there's a slew of example KQL queries available to access in the Logs blade itself. Just tap or click the Queries button to gain access.



## Exporting Queries

The Export option in the UI gives you the ability to export the query results in several ways.

You can export all data to a csv, export only the data in the displayed results, generate an M query for use in creating a Power BI dashboard, and export and open immediately in Microsoft Excel.



## New Alert Rule

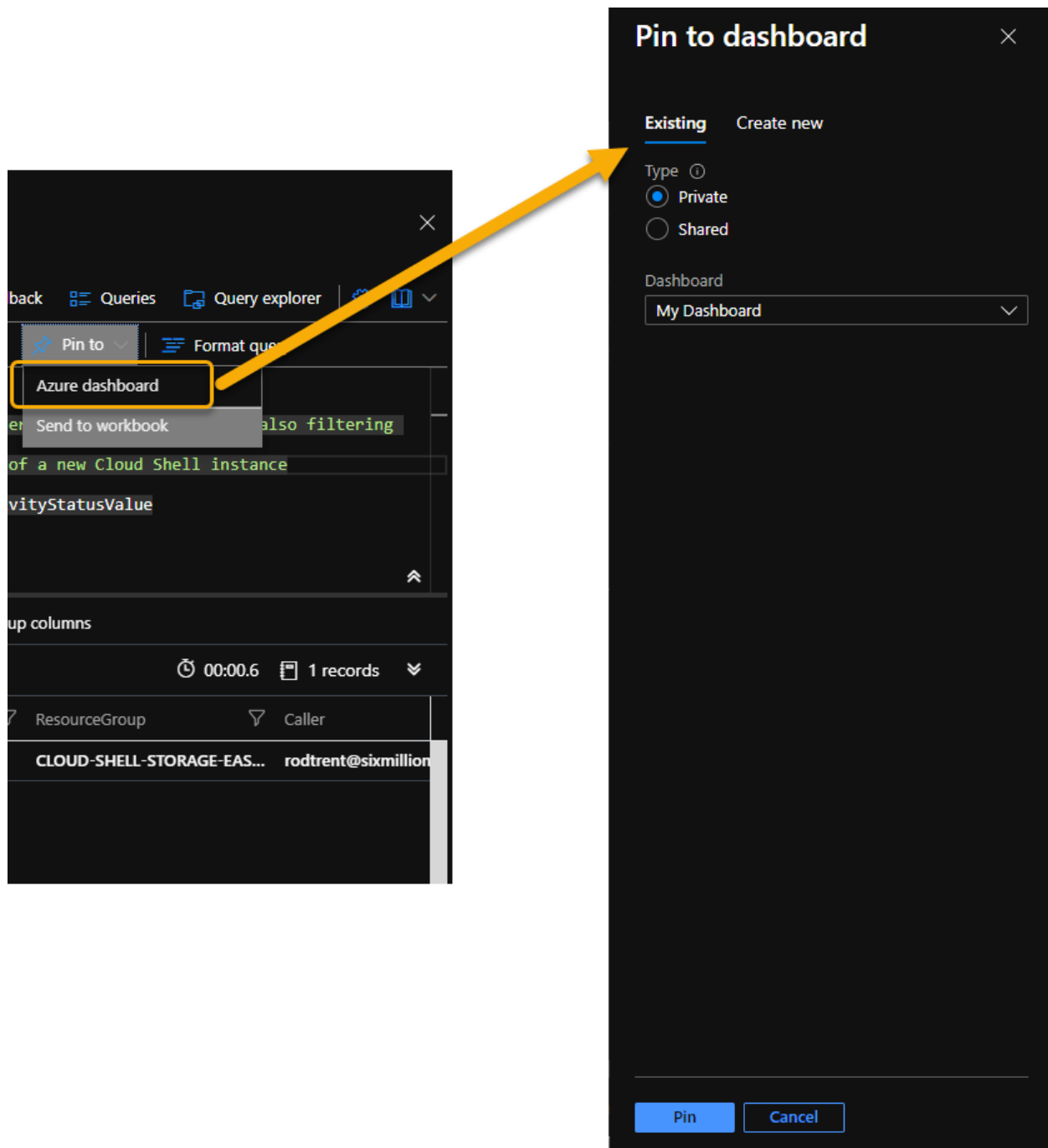
You can create rules for either Azure Monitor or Microsoft Sentinel directly from the Logs UI. This is an awesome feature that allows you to create and tune your query until it's perfect and then begin the steps to turn it into a rule to automatically analyze security for your environment. We're not quite at that step in this series, so we'll come back to this feature in Part 21. (See the [TOC](#))



## Pin to Dashboard or Workbook

Pin to Dashboard is an interesting feature in that you can take the query results that are formatted as a chart and pin the visualization directly to the standard

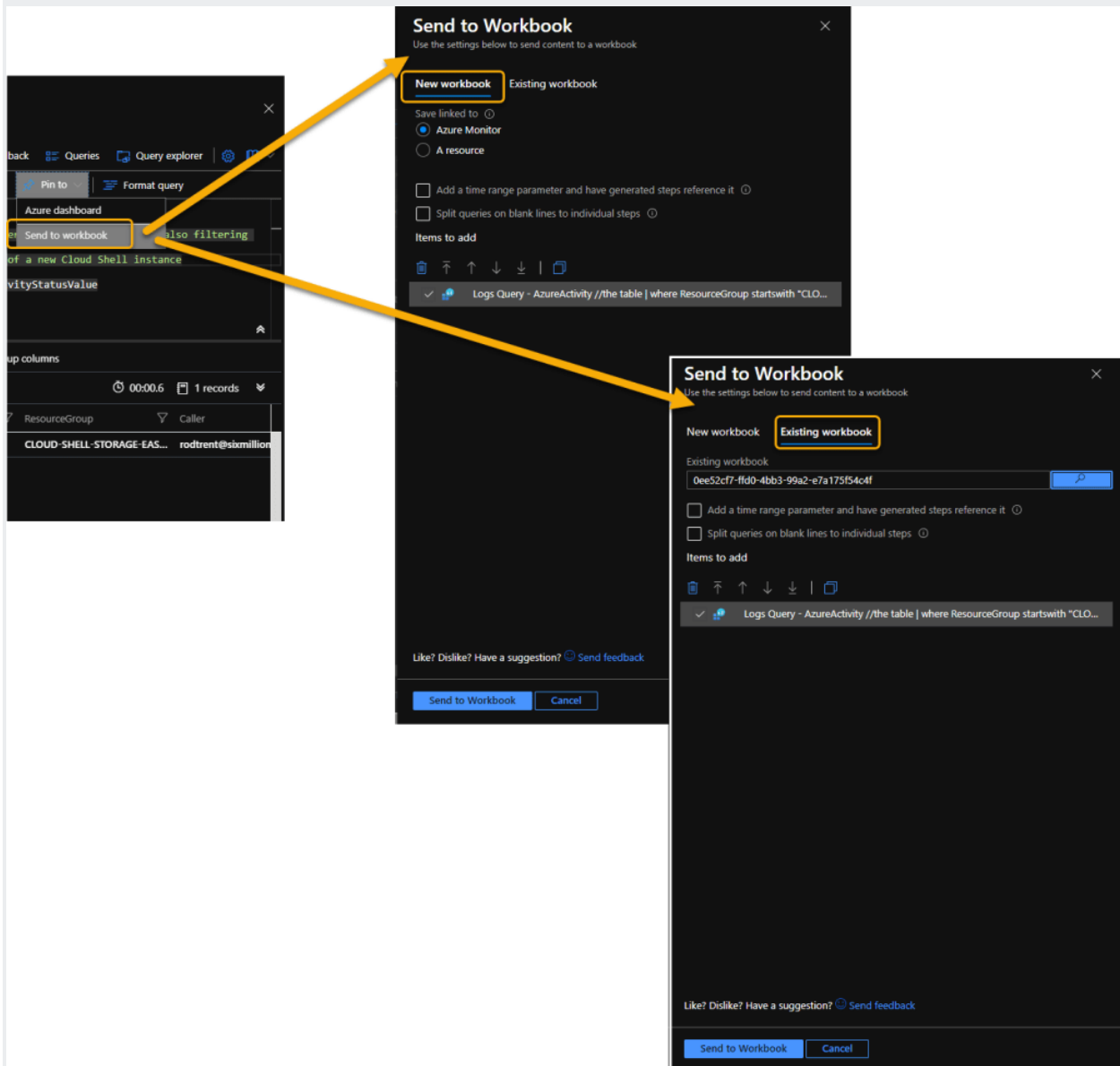
Azure portal dashboard. This dashboard can be your own private collection of visualizations or a collection that is shared among your teammates or even supplied so your manager has purview into operations.



Pin do Azure Dashboard

An additional option here is the ability to send the query to a Workbook. This is useful when you need to use the Logs blade to develop the query and instead of

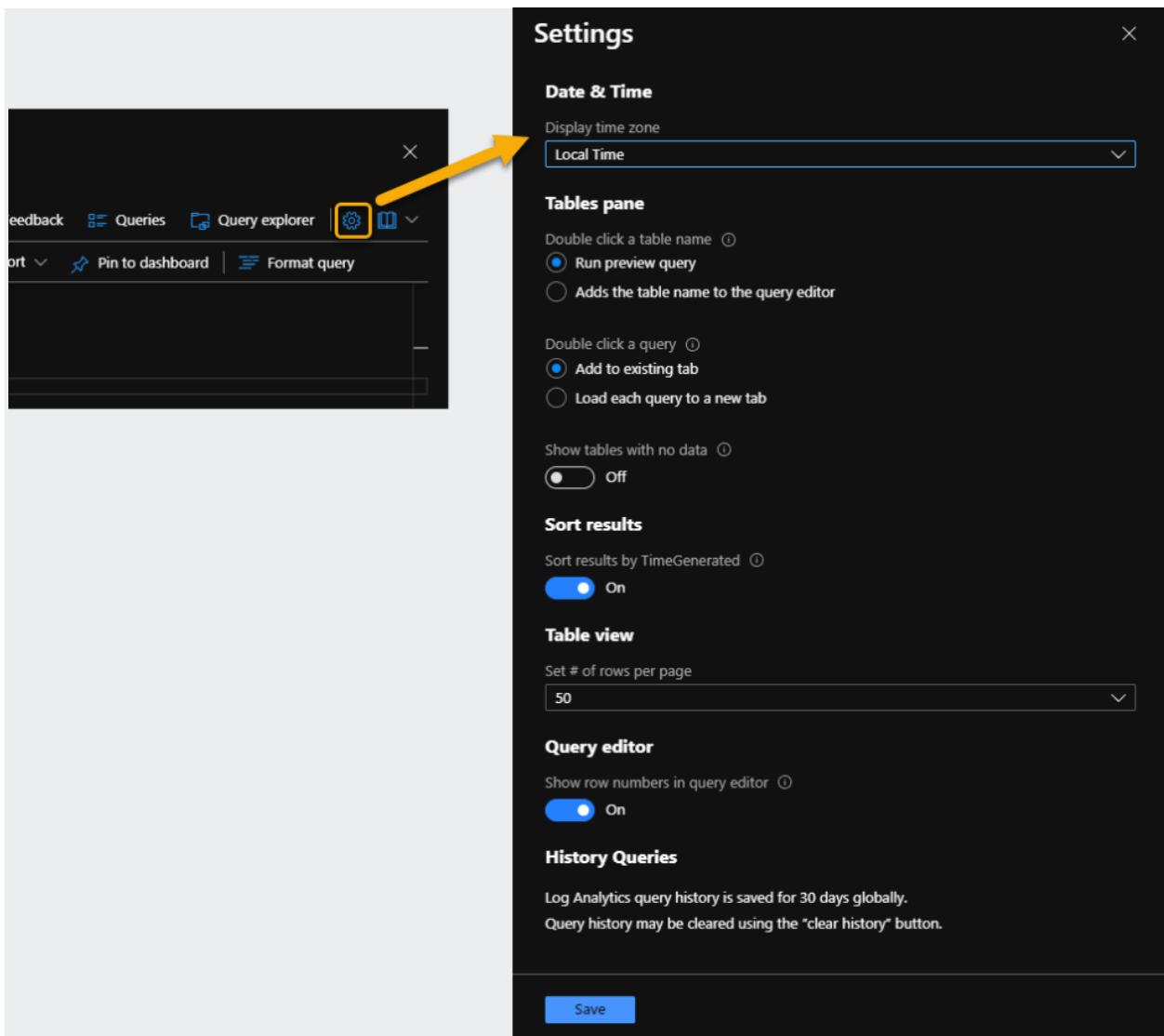
copying/pasting into an existing Workbook that you're currently developing, you can just insert it into the Workbook. Of course, you can create Workbooks from here using the new queries, too.



Send to a Workbook

## Settings

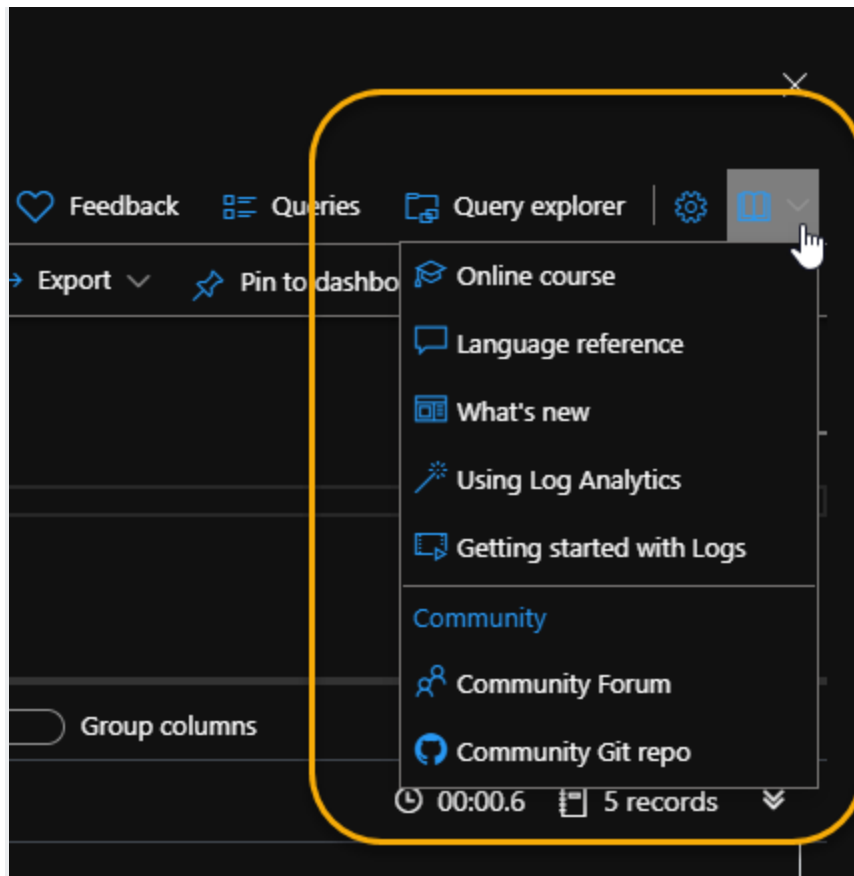
I'm not going to dig into each option, but the Settings icon contains configuration adjustments including things like how double-clicking works, if you want to see tables that contain no data, how many rows per page should display by default in the results window, and other things.



## In-UI Reference

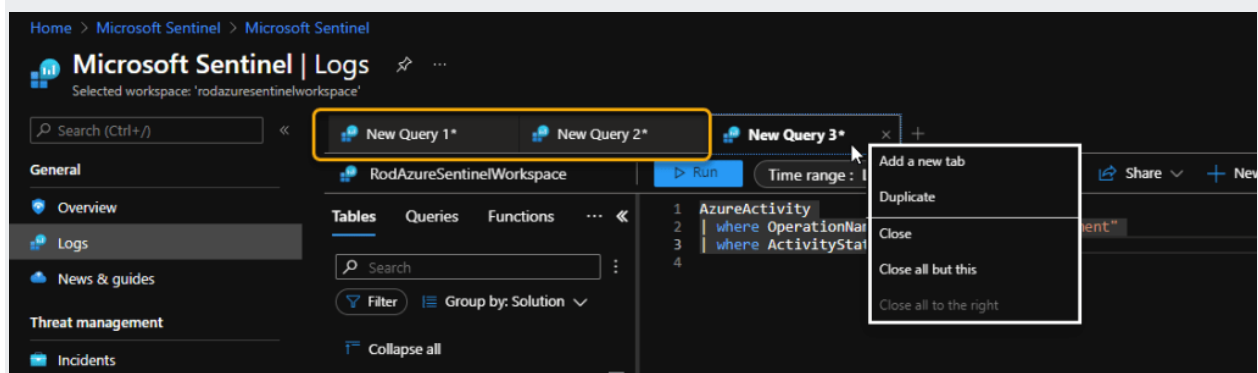
Lastly, to round out this intimate review of the Logs UI, there's a very good, very solid collection of references built into the UI. Some of those I've already supplied as references in [Part 1](#), but, like everything in Azure, this is also updated continually. So, keep an eye out here for updates.





## Tabs

To help keep you organized, much like a web browser the UI also supports tabs.

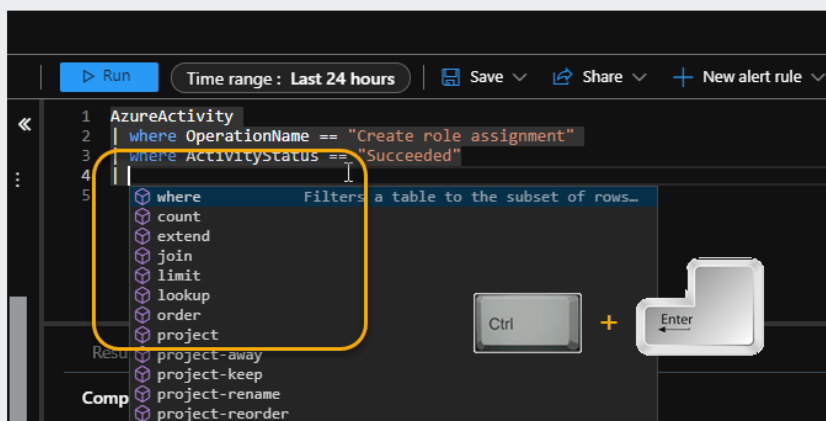
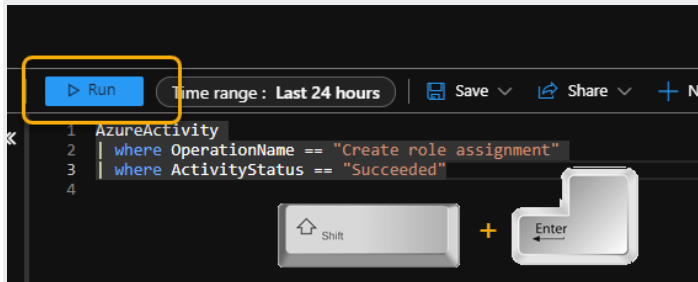


## Tabbing

This is awesome functionality to allow you to work on different queries or different datasets in each tab. If you right-click on each tab, there's a context menu pop-up that allows you manage the tabs in various ways including duplicating the current query in a new tab.

## Keyboarding Shortcuts

If you're a die-hard keyboarding fan like myself, rest easy knowing that you can help speed up your query development using a couple key combinations. It's also for us lazy people who can't suffer the time to lift our hands from the keyboard to locate the mouse and click on one of its buttons.

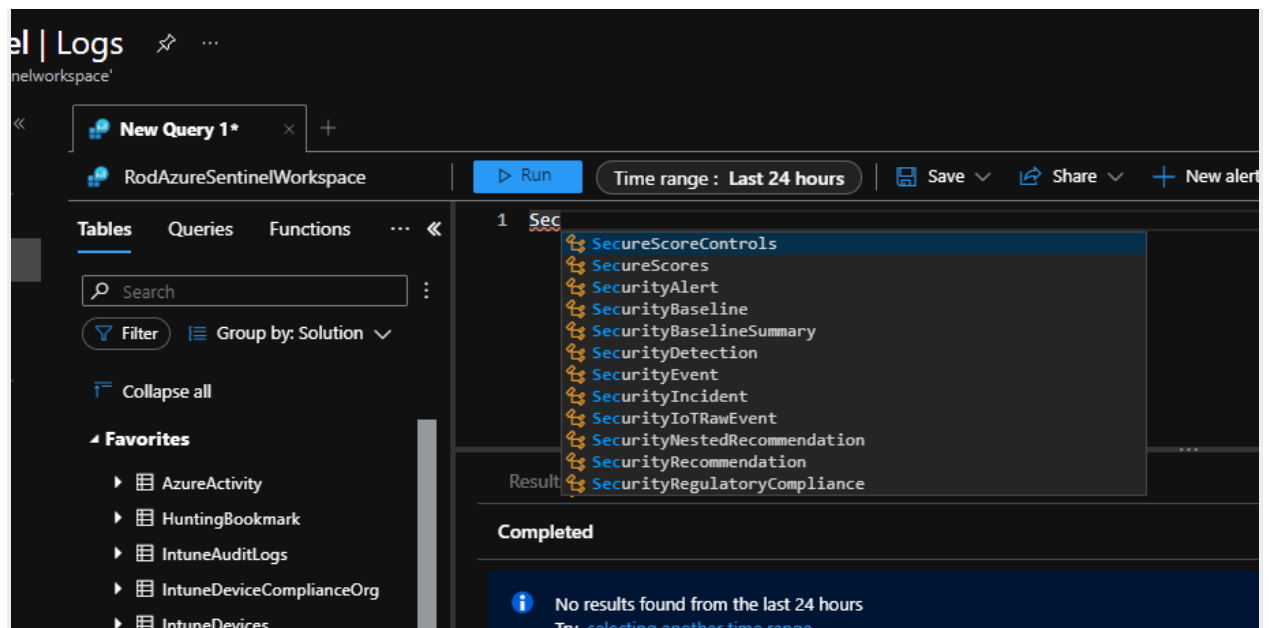


Keyboard shortcuts

**Shift + Enter** causes the query to run. **Ctrl + Enter** starts a new command line, complete with the command (*pipe* (|)) character.

## Intellisense for the Win

Much like how addressing an email works, the Logs UI will try everything it can to use autocomplete to try and figure out what it is you want to accomplish. Just start typing in the query area and the applicable options will display in a list.



But wait...there's more...

Next, in Part 7 (see the [TOC](#)), there's a bit more of the UI to talk about. But that deserves its own part since we'll be talking in relation to working with the tables and the schema.

## Must Learn KQL Part 7: Schema Talk

Before jumping directly into talking through some common KQL operators and providing you example queries for hands-on learning (*see the [TOC](#)*) in the next part of this series, there's some lingering discussion from [the last post around the UI](#), but also how this relates to table schema. I wanted to keep this information separate from the rest and in its own area because it will help you determine where things exist in the tables and how to better pinpoint the data. You saw in [Part 4](#) that it's easy to find anything in the data. But as you start getting closer and closer to taking the knowledge to develop your very own Analytics Rules for Microsoft Sentinel, you want to take the learning from [Part 5](#) and go just a tad bit further. This where an understanding of the schema becomes important.

The table schema is important. As with any data storage function or service, data is collected and stored – *most times appropriately* – in organized columns. I noted in [Part 5](#) about the [getschema operator](#) for KQL that produces the list of all columns and their types.

In case you missed it, or you forgot...

### Example:

```
OfficeActivity
```

```
| getschema
```

### Sample results:

Run Time range: Last 24 hours Save Share + New alert rule Export Pin to dashboard Format query

```
1 OfficeActivity
2 | getschema
```

Results Chart Columns Add bookmark Display time (UTC-05:00) Group columns

Completed. Showing results from the last 24 hours. 00:00.7 132 records

Column	ColumnName	ColumnOrdinal	DataType	ColumnType
>	TenantId	0	System.String	string
>	Application	1	System.String	string
>	UserDomain	2	System.String	string
>	UserAgent	3	System.String	string
>	RecordType	4	System.String	string
>	TimeGenerated	5	System.DateTime	datetime
>	Operation	6	System.String	string
>	OrganizationId	7	System.String	string
>	OrganizationId_	8	System.String	string
>	UserType	9	System.String	string
>	UserKey	10	System.String	string
>	OfficeWorkload	11	System.String	string

Page 1 of 3 50 items per page 1 - 50 of 132 items

Results from *getschema*

As you can see in the results, *getschema* shows a lot of great information. It shows the actual column names that are important to know for what types of information can be found, but also note the *DataType* and *ColumnType* results. These tell us how to query the data – or, rather, the approach we need to take (the type of KQL operator) to query, extract, and manipulate the data.

Using just the information displayed in the screenshot example, I can see that I can use [Part 5](#)'s knowledge to show regular Exchange users that sent emails. The following example shows that.

```
OfficeActivity
| where UserType == "Regular"
| where OfficeWorkload == "Exchange"
| where Operation == "Send"
```

```
| project UserId, UserDomain
```

Query example is located at: <https://cda.ms/3pf>

Note that not everything is as neatly stored and defined as the OfficeActivity table in the screenshot. I said earlier that *most times* data is stored neatly and orderly. There are exceptions and you need to be aware of these. In these cases, you'll need to utilize some parsing functions of KQL to extract the data yourself. But let's not focus on that here in this post. I promise, I'll dig into that later in the series (see the [TOC](#)) just before creating your first Analytics Rule.

But fortunately, *most times* data is store neatly and orderly. This is where the Data Connectors come into play in Microsoft Sentinel. The parsing is done for you when an actual Data Connector is in play. The "parser" is part of the Data Connector or the Sentinel Solution. For those situations where an official Data Connector does not exist, you may be called on to create your own parser. Again, I'll cover this later in this series, but I do want to call this out, as its important. So, for your efforts as you begin building your KQL knowledge, stick with the tables that are part of a Data Connector, otherwise you'll bump off into unknown territory that can get miry fast.

OK...with this knowledge firmly in-hand, let's jump back to the UI to talk about some areas in the console that help shortcut some of this activity.

## Column Types

As shown in the screenshot example, there are various KQL column types. Again, knowing these date column types will alter your approach for querying specific columns. I don't want to spend a lot of time here on this as to not start the varying levels of confusion. But I'll include this here so I can refer to it later on in the series.

The KQL column types are...

- **Basic**

- **int, long** (numerical types)
- **bool**: true, false (logical operators)
- **string**: "example", 'example'

- **Time**

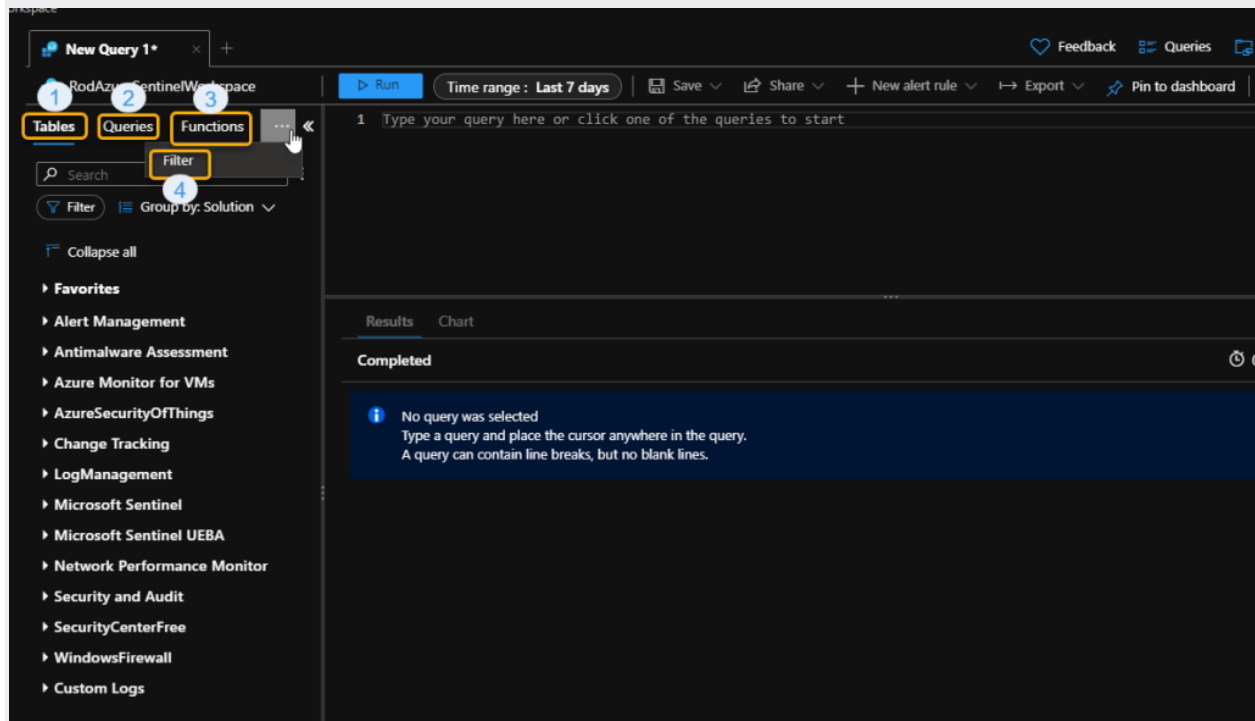
- **datetime**: datetime(2016-11-20 22:30:15.4), now(), ago(4d)
- **timespan**: 2d, 20m, time(1.13:20:05.10), 100ms

- **Complex**
- **dynamic:** JSON format

For anyone that's worked with any query language or data format before, these are not uncommon or new. As I talked about in [Part 2](#), KQL – the query language – was not designed to be difficult nor revolutionary. The revolutionary part is how it utilizes the power of the cloud (Azure) to accomplish sifting through mass seas of data quickly and efficiently. No, KQL – the query language – takes the best pieces of a lot of existing query languages. For example, anyone that's worked with SQL Server, will have an easy time with KQL.

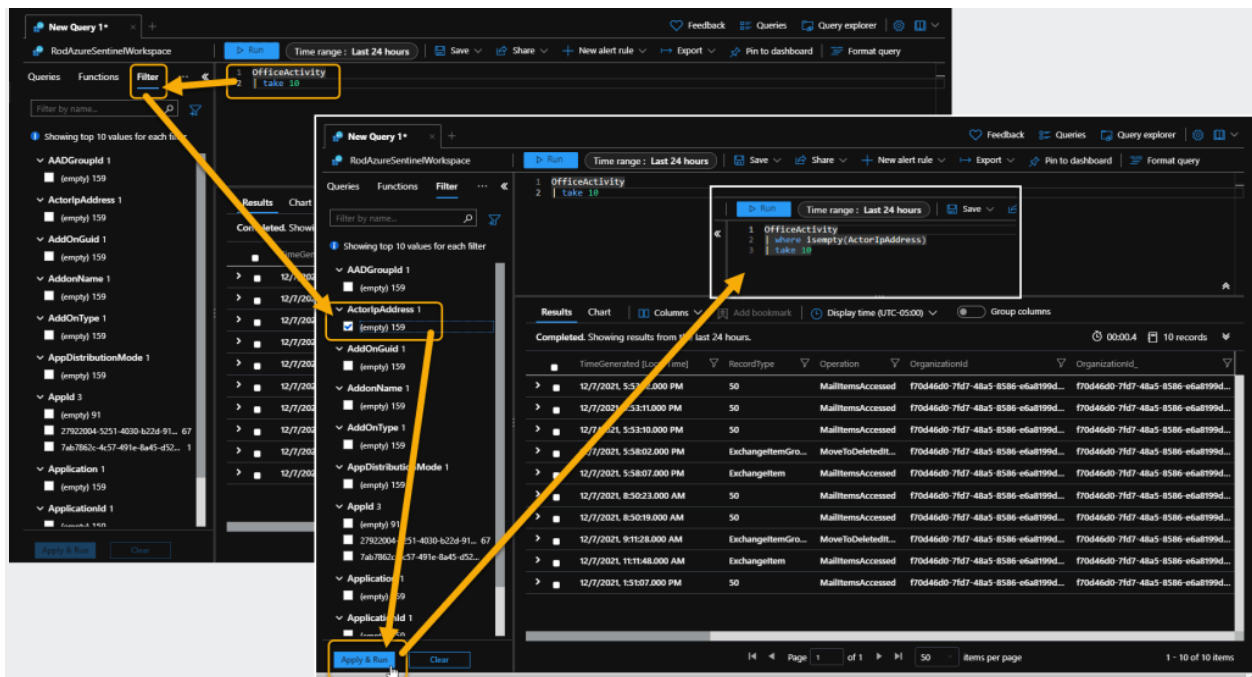
## Back to the UI

The UI has an area that aids in organizing and customizing the table/schema view, but it also has capabilities to enable easier and quicker access to KQL query creation. In this post, I'm not going to focus heavily on areas 2-4. You should be able to figure out how to click through and use most of those on your own. And, while I'll provide a quick overview of all the areas just now, I'll circle back and focus on the Tables area. As you're getting started learning KQL, this is the important area that will save you a lot of time learning to create your own queries.



## UI Overview:

1. This is the *Tables* list. This is where you can find all the available tables for which you can create queries against. We'll focus on this area just below.
2. This is the *Queries* list. This tab area contains a slew of pre-made KQL queries that you can spend hours and days executing, reverse engineering, and all other matters of query learning importance. These are separated by category types like Applications, Audit, Azure Monitor, Azure Resources, Containers, Databases, Desktop Analytics, IT & Management Tools, Network, Security, Virtual Machines, Windows Virtual Desktop, Workloads, and Others.
3. This is the *Functions* list. A *Function* is like a *stored procedure* in SQL, except in our case the query code is in KQL. This is a hugely useful component of KQL. I'll cover this in-depth later in the series (see the [TOC](#)). Did you know that the Watchlist feature of Microsoft Sentinel relies heavily on a Function? If you access the Function tab in the UI, you'll see the `_GetWatchlist` function.
4. The *Filter* tab. The Filter tab is absolutely awesome and delivers another shortcut method of developing your KQL queries. After running a query, the Filter tab will contain a list of empty data columns that you can select to filter out of the query results. Once a column is selected and applied, you can see in the screenshot that the query is updated automatically with the `where` operator to use as the filter mechanism and then the query is rerun. The `isempty()` component is used, which, in itself is a powerful tool that we'll talk about later in this series.

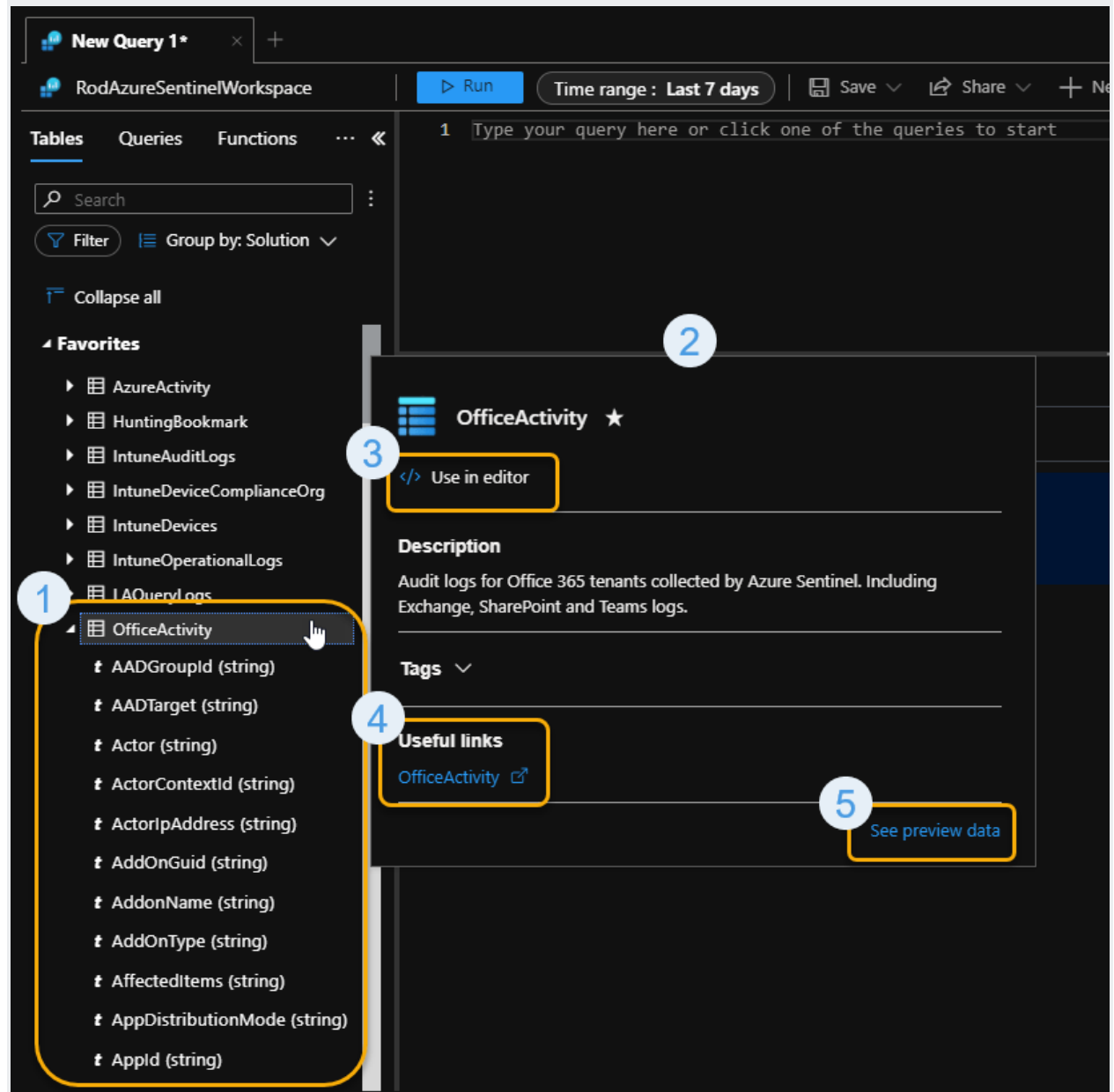


Filter tab



## Schema Area Focus

I noted in [Part 6](#) that everything that can be done in the UI we should eventually accomplish in the KQL query itself. That's still the case here, but the UI provides some neat shortcuts that shouldn't be overlooked.



2. Secondly, if you hover your mouse cursor over a Table name, a new pop-up window displays that provides even more query shortcut value. Also of importance, notice that the pop-up will display the description of the table.
3. If you click the *Use in editor* option, the Table name will automatically be placed in the query window so you can start querying against the table.
4. The *Useful links* option links directly to the [Azure Monitor Logs table reference](#) that I provided as a resource in [Part 1](#).
5. And, finally, the most excellent, super-cool shortcut is the capability to click and look at sample results from the table itself. Clicking on this will produce its own window like the following:

OfficeActivity ★

Use in editor

Description

Audit logs for Office 365 tenants collected by Azure Sentinel. Including Exchange, SharePoint and Teams logs.

Tags

Useful links

OfficeActivity

Preview Data

TenantId	RecordType	TimeGenerated [Local Time]	Operation	OrganizationId	Org:
e73fcae6-0260-4da5-9d56-f9e36...	50	12/7/2021, 7:56:04.000 AM	MailItemsAcc...	f70d46d0-7fd7-48a5-8586-e6a81...	f70d46
e73fcae6-0260-4da5-9d56-f9e36...	50	12/7/2021, 7:56:04.000 AM	MailItemsAcc...	f70d46d0-7fd7-48a5-8586-e6a81...	f70d46
e73fcae6-0260-4da5-9d56-f9e36...	50	12/7/2021, 7:56:04.000 AM	MailItemsAcc...	f70d46d0-7fd7-48a5-8586-e6a81...	f70d46
e73fcae6-0260-4da5-9d56-f9e36...	50	12/7/2021, 7:56:04.000 AM	MailItemsAcc...	f70d46d0-7fd7-48a5-8586-e6a81...	f70d46
e73fcae6-0260-4da5-9d56-f9e36...	50	12/7/2021, 7:56:04.000 AM	MailItemsAcc...	f70d46d0-7fd7-48a5-8586-e6a81...	f70d46
e73fcae6-0260-4da5-9d56-f9e36...	50	12/7/2021, 7:56:04.000 AM	MailItemsAcc...	f70d46d0-7fd7-48a5-8586-e6a81...	f70d46
e73fcae6-0260-4da5-9d56-f9e36...	50	12/7/2021, 7:56:04.000 AM	MailItemsAcc...	f70d46d0-7fd7-48a5-8586-e6a81...	f70d46
e73fcae6-0260-4da5-9d56-f9e36...	50	12/7/2021, 7:56:04.000 AM	MailItemsAcc...	f70d46d0-7fd7-48a5-8586-e6a81...	f70d46
e73fcae6-0260-4da5-9d56-f9e36...	50	12/7/2021, 7:56:04.000 AM	MailItemsAcc...	f70d46d0-7fd7-48a5-8586-e6a81...	f70d46
e73fcae6-0260-4da5-9d56-f9e36...	50	12/7/2021, 7:56:04.000 AM	MailItemsAcc...	f70d46d0-7fd7-48a5-8586-e6a81...	f70d46

Page 1 of 1

50 items per page

1 - 10 of 10 items

## Data Sampling

Incidentally, this most excellent, super-cool shortcut is actually a KQL query itself that uses the [take operator](#) that I'll cover later in the series. In fact, it's a take 10 similar to the following:

```
OfficeActivity
```

```
| take 10
```

This tells the query engine to display a random set of 10 records as a data sample. Because its random, every time it runs different data will display.

OK, now that we have all the concepts and UI functionality finally out of the way, it's time to start building queries using the most common KQL operators. From this point on in the series, I'll supply a KQL example based on an operator you can expect to use and see constantly in Microsoft Sentinel and our other security platform services. You should make it your intent to make use of the public KQL Playground I supplied in the Part 1 resources, or your own environment, to get hands-on with each operator I talk about.

You'll see as I go along, I'll take a simple query and start to build on it with each new part in this series. We'll begin simple and end up with a pretty interesting, but more complex query than what we started with.

# Must Learn KQL Part 8: The Where Operator

## Hands-on Recommendations



Before jumping directly into coverage of the first KQL operator, I want to extend some recommendations on how to proceed to ensure you get the most out of the hands-on opportunities through the remainder of this series.

In each new part of this series, I'll talk about a specific KQL operator, command, or concept and supply example queries that you can use to get hands-on experience. The examples will be available here in the text, but also in the Examples folder of the GitHub repository for this series (<https://aka.ms/MustLearnKQL>).

**Recommendation 1:** I know it will be tempting to just copy, paste, and run my query examples. But do yourself a favor and type them out instead. Use the blog page, or the book, as a side reference, and type out the queries character-by-character and line-by-line. I'm a big believer of learning by doing. Typing the queries out will solidify your new knowledge.

**Recommendation 2:** Consider using the KQL Playground (<https://aka.ms/LADemo>) from the [Part 1](#) resources as your learning environment when typing out the queries. The KQL Playground contains several data connections that you may not have in your own environment. The examples that I provide will have been tested to work and to show results. There's nothing more frustrating than being given an example and there are no results for your effort. You'll immediately start to think you did something wrong or that the query itself is bad. So, please, if possible, use the KQL Playground.

With that, let's jump into the first KQL operator...

## Where Operator

Bear with me (and *forgive* me) while I repeat myself. In part [Part 5: Turn Search into Workflow](#), I said the following...

*Filtering the data is the key to everything. <= Read that again. Filtering the data that is returned produces exact, actionable data. It also improves the results performance of our queries. Where the search operator may return thousands of rows of data in 15 seconds (or less), by properly filtering the data to return exactly what is necessary returns just the number of rows of data we asked for which greatly improves the processing time. Where the search operator may have taken 15 seconds, our new Search structure query will take 5 seconds or less. The Where operator is the key to this operation. Learn it. Know it. Keep the Where operator reference page handy: <https://cda.ms/3jh>.*

### Rod Trent, circa Part 5 of the Must Learn KQL series

That still holds true. So, based on that, would you agree with me that that makes this Part 8 one of the most important in the series? You betcha.

The syntax for the where operator will always be the same. Using our knowledge from [Part 3 on workflow](#), you know that the flow of the query needs to follow a logical path. We need to tell the query engine the table we want to query against, then we need to tell it how to filter that data.

### Where operator syntax:

TabLeName

| where predicate

### Allowable predicates:

- **String predicates:** ==, has, contains, startswith, endswith, matches regex, etc
- **Numeric/Date predicates:** ==, !=, <, >, <=, >=
- **Empty predicates:** isempty(), notempty(), isnull(), notnull()

## Where operator example:

In the following example, I've added the commenting character (*the double-forwardslash covered in [Part 3](#)*) to each line to explain what it is accomplishing.

```
SecurityEvent // The table
```

```
| where TimeGenerated > ago(1h) // Activity in the last hour
```

```
| where EventID == 4624 // Successful login
```

```
| where AccountType =~ "user" // case insensitive
```

As shown, the example queries the SecurityEvent table, looking for normal users (non-admins) that had a successful login in the last hour. Can you see that? For each command line (*separated by the pipe character (|) I talked about in [Part 3](#)*) the where operator is enacting on the data in a specific way based on the predicate. In the example, I've used the where operator three different times to further filter the results that will be produced. I can use the where operator ad nauseam, until the results are exactly what I need them to be.

Your results in the KQL Playground (<https://aka.ms/LADemo>) will look something like the following.

The screenshot shows the Azure Security Center KQL Playground interface. The query editor on the left contains the following KQL query:

```
1 SecurityEvent //The table
2 | where TimeGenerated > ago(1h) //Activity in the last hour
3 | where EventID == 4624 // Successful login
4 | where AccountType =~ "user" // case insensitive
5
6
```

The results pane on the right displays a table with 364 records. The table has the following columns: TimeGenerated [UTC], Account, AccountType, Computer, EventSourceName, Channel, Level, and EventID. The first row of data is highlighted, showing a successful login event for a user account on a DC10 server.

TimeGenerated [UTC]	Account	AccountType	Computer	EventSourceName	Channel	Level	EventID
12/8/2021, 3:42:02.850 PM	NA.CONTOSO\HOTELS.COM\jima...	User	DC10.na.contosohotels.c...	Microsoft-Windows-Security-Aud...	Security	8	4624
12/8/2021, 3:42:02.860 PM	NA.CONTOSO\HOTELS.COM\jima...	User	DC10.na.contosohotels.c...	Microsoft-Windows-Security-Aud...	Security	8	4624
12/8/2021, 3:42:03.370 PM	NA.CONTOSO\HOTELS.COM\jima...	User	DC10.na.contosohotels.c...	Microsoft-Windows-Security-Aud...	Security	8	4624
12/8/2021, 3:42:03.397 PM	NA.CONTOSO\HOTELS.COM\jima...	User	DC10.na.contosohotels.c...	Microsoft-Windows-Security-Aud...	Security	8	4624
12/8/2021, 3:42:03.413 PM	NA.CONTOSO\HOTELS.COM\jima...	User	DC10.na.contosohotels.c...	Microsoft-Windows-Security-Aud...	Security	8	4624
12/8/2021, 3:42:03.430 PM	NA.CONTOSO\HOTELS.COM\jima...	User	DC10.na.contosohotels.c...	Microsoft-Windows-Security-Aud...	Security	8	4624
12/8/2021, 3:42:03.440 PM	NA.CONTOSO\HOTELS.COM\jima...	User	DC10.na.contosohotels.c...	Microsoft-Windows-Security-Aud...	Security	8	4624
12/8/2021, 3:42:03.460 PM	NA.CONTOSO\HOTELS.COM\jima...	User	DC10.na.contosohotels.c...	Microsoft-Windows-Security-Aud...	Security	8	4624
12/8/2021, 3:42:03.497 PM	NA.CONTOSO\HOTELS.COM\jima...	User	DC10.na.contosohotels.c...	Microsoft-Windows-Security-Aud...	Security	8	4624
12/8/2021, 3:42:03.507 PM	NA.CONTOSO\HOTELS.COM\jima...	User	DC10.na.contosohotels.c...	Microsoft-Windows-Security-Aud...	Security	8	4624
12/8/2021, 3:42:03.513 PM	NA.CONTOSO\HOTELS.COM\jima...	User	DC10.na.contosohotels.c...	Microsoft-Windows-Security-Aud...	Security	8	4624
12/8/2021, 3:42:03.523 PM	NA.CONTOSO\HOTELS.COM\jima...	User	DC10.na.contosohotels.c...	Microsoft-Windows-Security-Aud...	Security	8	4624

I'm keeping it simple here and focusing only on the string and time predicates. As we move on in the series, we'll get to the other predicates.

**EXTRA:** There is one additional piece of clarification I need to make. In the third (*last line*) where statement of the example query there's an interesting looking predicate (`=~`). The tilde (`~`) character can be used in string predicates to cause the query engine to ignore case (*case insensitivity*). So, for our example, I'm telling the query engine to find every occurrence of the word "user" in the AccountType column no matter if it's spelled "User" or "user" or "uSEr", etc. Otherwise, it's going to return my request verbatim which could result in zero results for the AccountType column.

Here, try it yourself in the KQL Playground (<https://aka.ms/LADemo>) without the tilde and notice that the AccountType column is empty:

SecurityEvent // The table
where TimeGenerated > ago(1h) // Activity in the last hour
where EventID == 4624 // Successful logon
where AccountType == "user" // case sensitive

The tilde is an extremely useful tool particularly if there have been data or schema changes.



**EXTRA CREDIT:** If you're hungry for more of the where operator, and just want to continue building your KQL knowledge until the next part in this series (see the [TOC](#)), take the original query example to the KQL Playground (<https://aka.ms/LADemo>) and run it line-by-line to see how each line changes the results. You can insert and remove the double-forwardslash (`//`) character at the beginning of each command line to comment it out or to include it.

For example, the following query will show more data than just in the last hour because, as you can see, the TimeGenerated filter line with the double-forwardslash character.

```
SecurityEvent // The table
```

```
// | where TimeGenerated > ago(1h) // Activity in the last hour
```

```
| where EventID == 4624 // Successful logon
```

```
| where AccountType =~ "user" // case insensitive
```



# Must Learn KQL Part 9: The Limit and Take Operators

Because *limit* and *take* are so similar and used for the same purposes, I'm going to combine those in this part of this series. I'm not going to rehash my hands-on recommendations here, but please check out the section in [Part 8](#) for those if you either missed it or have forgotten. In my opinion, the hands-on part of this series is the most important piece.

Up front – there are *no functional* differences between *limit* and *take*. They're like fraternal twins. They have the same origin and similar attributes but have different names and looks.

In some cases, there are those KQL operators or commands that have similar functions, but one is better than another in how it reacts with the underlying technologies. Or, better said, one is better performing in most situations than another. In fact, we have a living document around this. See the [KQL Best Practices doc](#) for more information. Take special notice of the *has* and *contains* operators in the list in the Best Practices doc since I talked about the String Predicates in [Part 8](#).

That said, since there are no true functional differences between *limit* and *take* it comes down to personal preference.

## Limit/Take operator syntax:

Tablename

| *limit* <number>

-or-

Tablename

| *take* <number>

There are a few things to keep in mind about these fraternal twin operators:

- **Sort is not guaranteed to be preserved.** This speaks for itself. Don't expect any special sorting of columns of data to work.
- **Consistent result is not guaranteed.** No matter how many times you run the same query with *limit* or *take*, it will most assuredly produce different results. The results are always random.
- **Very useful when trying out new queries or performing data sampling.** *Data Sampling* is a powerful capability of any data scientist or meager KQL query maven. This is a similar activity for when we used the *search* operator in [Part 4](#).
- **Default limit is 30,000.** No matter what number you supply in the query, the results will never show more than 30,000. That's a hard limit. And, when you think about it, since *limit* and *take* are part of a data sampling technique, you may want to seriously rethink your strategy (*and use a different operator*) if you need more than 1,000 rows of data returned – and that's a generous number.

### limit/take operator example:

As recommended in [Part 8](#), use the KQL Playground (<https://aka.ms/LADemo>) to test the following query example. And for those wanting to better retain the knowledge, try typing the query out instead of copying/pasting.

And guess what? I've supplied **both** the *limit* and *take* operator versions so you can start to formulate your favorite.

```
SecurityEvent // The table
```

```
| where TimeGenerated > ago(1h) // Activity in the last hour
```

```
| where EventID == 4624 // Successful logon
```

```
| where AccountType =~ "user" // case insensitive
```

```
| limit 10 //random data sample or 10 records
```

-or-

```
SecurityEvent // The table
```

```
| where TimeGenerated > ago(1h) // Activity in the last hour
```

```
| where EventID == 4624 // Successful logon
```

```
| where AccountType =~ "user" // case insensitive
```

```
| take 10 //random data sample or 10 records
```

Also notice that I'm using the same query example from [Part 8](#) – just adding the *limit* and *take* command lines at the end. I'll use this same query throughout (as much as possible) to show a standard method of query development that will lead to creating your very first Analytics Rule for Microsoft Sentinel. Creating an Analytics Rule for Microsoft Sentinel is a very similar process of starting simple and building bigger.

Your results for either query example will look like the following. Just remember that your results will be slightly different because of the random nature of the operators.

The screenshot shows the Microsoft Sentinel interface. The query editor at the top contains the following KQL query:

```
1 SecurityEvent // The table
2 | where TimeGenerated > ago(1h) // Activity in the last hour
3 | where EventID == 4624 // Successful login
4 | where AccountType =~ "user" // case insensitive
5 | limit 10 //random data sample or 10 records
6
7 -or-
8
9 SecurityEvent // The table
10 | where TimeGenerated > ago(1h) // Activity in the last hour
11 | where EventID == 4624 // Successful login
12 | where AccountType =~ "user" // case insensitive
13 | take 10 //random data sample or 10 records
```

Below the query editor, the results table is displayed. The table has the following columns: TimeGenerated [UTC], Account, AccountType, Computer, EventSourceName, Channel, Task, Level, and EventID. The results show 10 records of successful login events for the user 'user' from the computer 'DC11.na.contoso-hotels.com'.

TimeGenerated [UTC]	Account	AccountType	Computer	EventSourceName	Channel	Task	Level	EventID
12/13/2021, 4:46:24.970 PM	NA.CONTOSOHOTELS.COM\jima...	User	DC11.na.contoso-hotels.com	Microsoft-Windows-Security-Aud...	Security	12.544	8	4.6
12/13/2021, 4:46:24.993 PM	NA.CONTOSOHOTELS.COM\jima...	User	DC11.na.contoso-hotels.com	Microsoft-Windows-Security-Aud...	Security	12.544	8	4.6
12/13/2021, 4:46:25.010 PM	NA.CONTOSOHOTELS.COM\jima...	User	DC11.na.contoso-hotels.com	Microsoft-Windows-Security-Aud...	Security	12.544	8	4.6
12/13/2021, 4:46:25.077 PM	NA.CONTOSOHOTELS.COM\jima...	User	DC11.na.contoso-hotels.com	Microsoft-Windows-Security-Aud...	Security	12.544	8	4.6
12/13/2021, 4:46:25.110 PM	NA.CONTOSOHOTELS.COM\jima...	User	DC11.na.contoso-hotels.com	Microsoft-Windows-Security-Aud...	Security	12.544	8	4.6
12/13/2021, 4:46:25.140 PM	NA.CONTOSOHOTELS.COM\jima...	User	DC11.na.contoso-hotels.com	Microsoft-Windows-Security-Aud...	Security	12.544	8	4.6
12/13/2021, 4:46:25.290 PM	NA.CONTOSOHOTELS.COM\jima...	User	DC11.na.contoso-hotels.com	Microsoft-Windows-Security-Aud...	Security	12.544	8	4.6
12/13/2021, 4:46:25.307 PM	NA.CONTOSOHOTELS.COM\jima...	User	DC11.na.contoso-hotels.com	Microsoft-Windows-Security-Aud...	Security	12.544	8	4.6
12/13/2021, 4:46:25.400 PM	NA.CONTOSOHOTELS.COM\jima...	User	DC11.na.contoso-hotels.com	Microsoft-Windows-Security-Aud...	Security	12.544	8	4.6
12/13/2021, 4:46:25.430 PM	NA.CONTOSOHOTELS.COM\jima...	User	DC11.na.contoso-hotels.com	Microsoft-Windows-Security-Aud...	Security	12.544	8	4.6

Randomness

## Must Learn KQL Part 10: The Count Operator

If you remember in the last part of this series ([Part 9 on limit and take operators](#)) I noted that in the query tool the query results are limited to 30,000 rows. Depending on how far back the data is being stored, i.e., your Log Analytics workspace retention settings, there might possibly be hundreds of thousands of rows of data in the tables. Now, going back to what I said before (*also in [Part 9](#)*), if you need more than 1,000 rows of data to determine if something exists or is impactful to the environment, you might want to change your strategy. In my opinion, just knowing that a potential security situation exists is important enough to circle the wagons.

But a *count* of something is a good measure to get a better understanding of overall impact of a situation.

For example, if there's one or two occurrences of a single person locking themselves out of their account in the last 30 days, that's not usually a big deal. It's most likely someone who forgot their password. As remediation, we can suggest to their manager that they might need to invest in training. We've all worked with those types of people. And as many of those types we know professionally, we probably know many more personally. My mom, my dad, my wife – yes, I'm also afflicted by those that believe passwords are just a nuisance and not something worth remembering.

But if we have record of that single person locking themselves out of their account 100 times in the last 30 days, that's a more immediate concern.

This is where the [\*count\*](#) operator really shines.

### Count operator syntax:

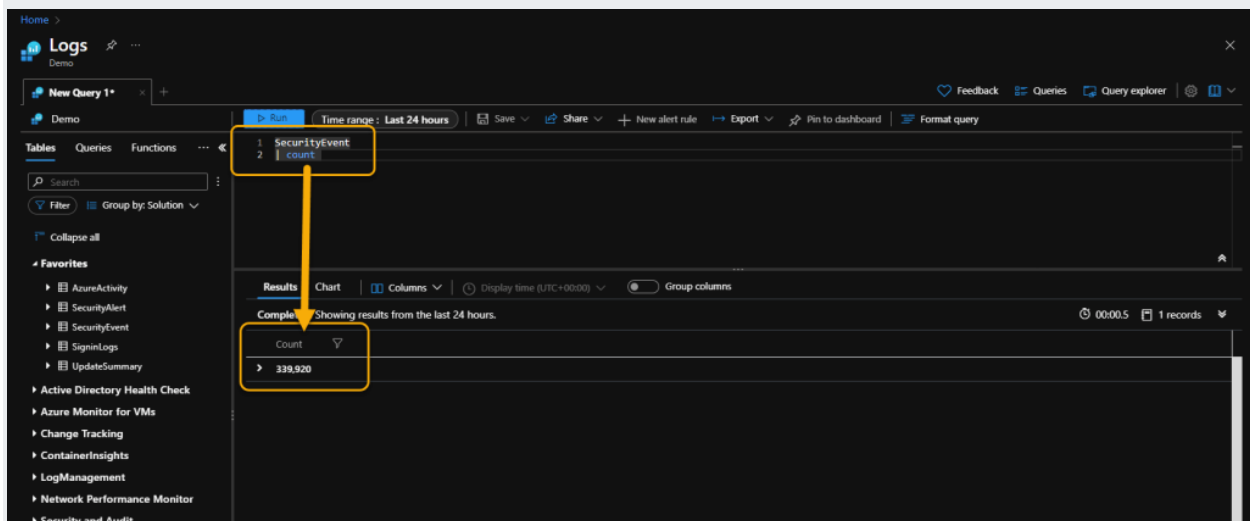
Tablename
<i>count</i>

On its own, just using the operator syntax listed above will show the exact number of rows in a given table. For example, the following query shows how many rows exist in the SecurityEvent table.

```
SecurityEvent
```

```
| count
```

Typing out this query in the KQL Playground (<https://aka.ms/LADemo>) will show something similar to the following screenshot...



Number of rows in the SecurityEvent table

Now, let's take the same query we've been using for all our query building exercises so far and add the count operator to it. Type this query in the KQL Playground (<https://aka.ms/LADemo>):

```
SecurityEvent // The table
```

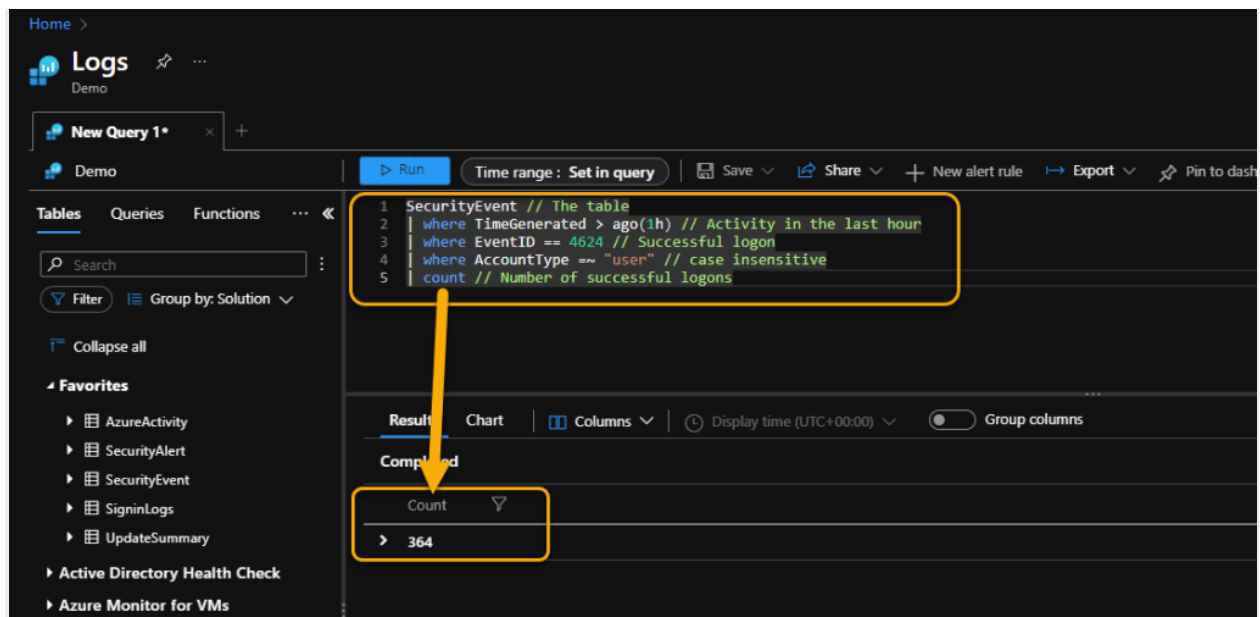
```
| where TimeGenerated > ago(1h) // Activity in the last hour
```

```
| where EventID == 4624 // Successful logon
```

```
| where AccountType =~ "user" // case insensitive
```

```
| count // Number of successful logons
```

As before, the query results show us the number of successful logons in the last hour by all standard (non-admin) users. But, now with the count operator, the results tell us the total number of times this occurred.



Number of successful logons

I think we can agree that this is much more impactful data than just showing row after row of data and then having to manually sift through it. It's important to grasp that adding a simple line to our original query changed everything. It made it even more powerful and even more relevant for our purposes.

Hopefully, you see as we are building our query toward Analytics Rule creation (see the [TOC](#)), that only simple steps are required to get us there. Each part of this series is intended as just one more simple step in the learning process.

The [count](#) operator will be a key to Analytic Rule development. In the next part of this series (see the [TOC](#)), I'll talk about the [summarize](#) operator where the [count](#) operator will come into play again. In fact, we'll be working with [count](#) quite a bit throughout the series. As important as the [where](#) operator is for filtering data ([Part 8](#)), the [count](#) operator is equally significant for its myriad of uses including helping create graphs and charts when we get to the [render](#) operator (see the [TOC](#)).



**EXTRA CREDIT:** The number of successful logons (Event ID 4624) is not necessarily something we look for when searching for security events. Instead, Event ID 4625 (unsuccessful logon) is the one most used to expose issues. For extra work and fun, in the KQL Playground (<https://aka.ms/LADemo>) simply change 4624 in the query to 4625 and run it again.

```
SecurityEvent // The table

| where TimeGenerated > ago(1h) // Activity in the last hour

| where EventID == 4625 // Unsuccessful logon

| where AccountType =~ "user" // case insensitive

| count // Number of successful logons
```

In the KQL Playground (<https://aka.ms/LADemo>) you should get 0 (zero) results for your effort, but this is also an impactful number. If there are no unsuccessful logons in your environment – ever – you have been gifted with the unicorn of end-user populations and you should never leave your post.

The screenshot shows the Azure KQL Playground interface. On the left, there's a sidebar with 'Home', 'Logs Demo', and a 'New Query 1\*' tab. Below this is a 'Tables' section with a search bar and a list of tables: AzureActivity, SecurityAlert, SecurityEvent, SigninLogs, UpdateSummary, Active Directory Health Check, and Azure Monitor for VMs. The main area displays a KQL query:

```
1 SecurityEvent // The table
2 | where TimeGenerated > ago(1h) // Activity in the last hour
3 | where EventID == 4625 // Unsuccessful logon
4 | where AccountType =~ "user" // case insensitive
5 | count // Number of successful logons
```

The query is highlighted with a yellow box. Below the query, there's a 'Run' button and a 'Time range: Set in query' dropdown. The results section shows a table with one column, 'Count', and one row with the value '0'. A yellow arrow points from the '4625' in the query to the '0' in the result.

Yay! No unsuccessful logons!

## Must Learn KQL Part 11: The Summarize Operator

For this part in this Must Learn KQL series, I once again want to take the logical next step as we march toward generating our very first Microsoft Sentinel Analytics Rule (see the [TOC](#) for the cadence). We have a lot of ground to cover before then, but the next few operators we talk about are useful for various reasons – one of those reasons, like this section’s [Summarize operator](#) talk, is to produce number data to encapsulate actions. By creating thresholds, we can generate additional logic for how we want to react to situations. For example, if there’s one person that failed login in the last 10 days, it’s a non-issue. But, if that account failed login 100 times in the last 5 minutes – well – we have a problem. Summarizing the data makes it more meaningful.

The [Summarize operator](#) does just what it suggests – it *summarizes* data. In deeper terms, it *produces a table (in the results) that aggregates the content of the input table*. As an example of this, use the following KQL query in the KQL Playground (<https://aka.ms/LADemo>) to see the results. And, as before, try typing the query into the KQL Playground instead of just a copy/paste operation. If you get an error, you might’ve fat-thumbed something so you can use the inline code to compare against. Query troubleshooting is a great skillset to have.

```
SecurityEvent // The input table
```

```
| where TimeGenerated > ago(1h) // Activity in the last hour
```

```
| where EventID == 4624 // Successful logon
```

```
| summarize count() by AccountType, Computer //Show the number of successful logons per
```

```
computer and what type of account is being used
```

Your results should be similar to the following:



Home >

**Logs**  
Demo

New Query 1\*

Run Time range: Set in query Save Share + New alert rule Export Pin to dashboard

1 SecurityEvent // The table  
2 | where TimeGenerated > ago(1h) // Activity in the last hour  
3 | where EventID == 4624 // Successful logon  
4 | summarize count() by AccountType, Computer //Show the number of successful logons per computer and what t

Results Chart Columns Display time (UTC+00:00) Group columns

Completed

AccountType	Computer	count_
> Machine	DC00.na.contosohotels.com	567
> Machine	DC01.na.contosohotels.com	342
> Machine	DC10.na.contosohotels.com	301
> Machine	DC11.na.contosohotels.com	299
> User	DC11.na.contosohotels.com	96
> User	DC01.na.contosohotels.com	112
> User	DC10.na.contosohotels.com	112
> User	SQL00.na.contosohotels.com	8
> User	DC00.na.contosohotels.com	15
> Machine	JBOX00	7
> Machine	Victim00.na.contosohotels....	9
> Machine	RETAILVM01	11
> Machine	AppBE00.na.contosohotels....	13

Page 1 of 1 50 items per page

## Summarize Operator Syntax

Tablename

| summarize *Aggregation* [by *Group Expression*]

- **Simple aggregation functions:** count(), sum(), avg(), min(), max(),
- **Advanced aggregation functions:** arg\_min(), arg\_max(), percentiles(), makelist(), countif()

The Simple aggregations should speak for themselves. While the Advanced ones may require a bit more information. I'll leave these descriptions here for posterity, but we'll actually circle back later in this series to cover them in depth. Again, our series is a building operation. I don't want to give you too much too soon and want to do so in a logical fashion so it all makes sense and learning is easier to retain.

Advanced aggregations:

- **arg\_min(), arg\_max():** *returns the extreme value*
- **percentiles():** *returns the value at the percentile*
- **make\_list(), make\_set():** *returns a list of all values/distinct values respectively*

Now, that we're deep into this series with Part 11, I'm going to attempt to do a bit less handholding. If this is your first introduction to this series, I highly suggest going back and making it through from the start because each new section or chapter builds on the previous ones. You can find the entire series tabulated in the [TOC](#).



With that, I want to leave you with some additional Summarize exercises that you can work with in the KQL Playground (<https://aka.ms/LADemo>). These use the Advanced aggregates and I'll refer back to these later.

```
SecurityEvent
```

```
| where EventID == 4624
```

```
| summarize arg_max(TimeGenerated, *) by Account
```

```
AzureDiagnostics
```

```
| summarize arg_max(TimeGenerated, *) by ResourceId
```

```
SecurityEvent
```

```
| summarize AdminSuccessfulLogons = countif(Account contains "Admin" and EventID == 4624),
```

```
AdminFailedLogons = countif(Account contains "Admin" and EventID == 4625)
```

As you can tell, this is not quite the end of our Summarize operator discussion. There will be plenty more. In fact, other than later in the series, I'll talk about Summarize even more in the very next part when I cover the Render operator in Part 12.

## Must Learn KQL Part 12: The Render Operator

This chapter may seem like somewhat of a detour on our path to using KQL queries to create Analytics Rules for Microsoft Sentinel, but there's some very real value in turning rows and columns of data into visualizations. Sure enough, across Microsoft Sentinel, you'll use KQL for almost everything – that includes the Workbooks feature that allows organizations to develop their own *views* of the security data.

Workbooks can be used to create dashboards of consolidated data. I've worked with several customers in 911-emergency-type settings where they've erected massive screens at the front of the room. KQL allows them to create the dashboard views that display on the screens so the entire security team can be privy to potentially nefarious operations in near real time. So, having an understanding of how KQL can produce visualizations is important.

There's another great reason to put some effort into learning how to transform static data into graphs and charts – and it's not just because I said in [Part 6](#) when I gave the tour of the User Interface...

*I preface this post by saying this: everything discussed in this post about the User Interface (UI) can be done (and should be done, eventually) in the KQL query itself.*

*Rod Trent, Part 6 of the Must Learn KQL series*

I like to think of myself as reader. I remember growing up reading book after book and loving it. One of my favorite series was the [Hardy Boys](#). I read the entire series and some of the books more than once. I seriously believe that mystery books like those have a lot to do with my fascination with cybersecurity. But, somewhere along the way I was introduced to comic books and telling stories with pictures and words was fantastic to me. I'm a visual, hands-on type of learner so comic books really filled a void.

Fast forward to today. Every day I read tomes of emails, Teams messages, social media posts, etc., etc. And frankly this has soured me to just general reading. Friends and family are always recommending books and I just look at them and

shake my head. Once they see my response to their recommendations, they quickly switch to “*well, just get the audiobook version, you’ll love it*” as if that’s somehow a better alternative. I truly wish I could go back and be that early reader and get excited about it all. But I read so much as part of my job, sitting down with a book in a quiet room now seems like torture to me. But, as a visual learner, that comic book style still appeals to me. If you can show me in a meaningful way the storyline of a threat, I’m all in.

*P.S. I still read comics books to this day.*

And that, for me, is where the KQL [Render operator](#) comes in. *Render* tells the query engine that you want to take the data you’ve supplied, and show it in any of the following ways (visualizations):

- areachart – Area graph. First column is the x-axis and should be a **numeric** column. Other **numeric** columns are y-axes.
- barchart – First column is the x-axis and can be **text, datetime or numeric**. Other columns are **numeric**, displayed as horizontal strips.
- columnchart – same as barchart.
- piechart – First column is color-axis, second column is **numeric**.
- scatterchart – Points graph. First column is the x-axis and should be a **numeric** column. Other **numeric** columns are y-axes.
- table – this is the default view.
- timechart – Line graph. First column is x-axis, and should be **datetime**. Other (**numeric**) columns are y-axes.

Something important to know is that each visualization requires a certain data type before it will display. I’ve **boldened** those requirements in the list above. As you see, many of the requirements are *numeric*, hence why I covered the [Summarize operator](#) in the previous part/chapter 11 ([see the TOC](#)). What I didn’t cover in Summarize was how to take numeric and datetime values and group them into smaller specific values for use in visual displays using the Render operator. In the examples below, I’m including bin and time for your hands-on exercises in the KQL Playground (<https://aka.ms/LADemo>) but understand that Bin rounds values down to an integer multiple of a given bin size. It’s used frequently in combination with *summarize by*. If you have a scattered set of values, they will be grouped into a smaller set of specific values. For the first examples below, recognize that *bin* is being used to split out a week’s worth of data into daily chunks.

## Render Operator Syntax

Tablename

| render visualization

## Just the Data

SecurityEvent //The table

| where TimeGenerated > ago(7d) //Looking at data in the last 7 days

| summarize count() by bin(TimeGenerated, 1d) //Using Bin to group the data by each day

### Completed

	TimeGenerated [UTC]	count_
>	1/10/2022, 12:00:00.000 AM	254,327
>	1/9/2022, 12:00:00.000 AM	343,903
>	1/6/2022, 12:00:00.000 AM	337,273
>	1/8/2022, 12:00:00.000 AM	324,150
>	1/7/2022, 12:00:00.000 AM	322,246
>	1/5/2022, 12:00:00.000 AM	338,209
>	1/4/2022, 12:00:00.000 AM	334,734
>	1/3/2022, 12:00:00.000 AM	84,785

Just the data

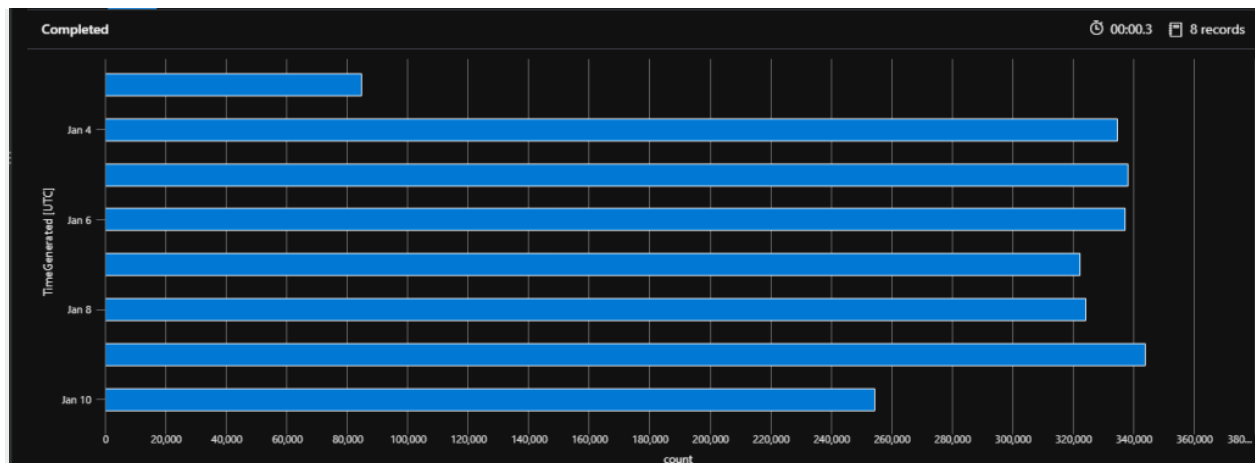
## The Data as a Barchart

SecurityEvent //The table

| where TimeGenerated > ago(7d) //Looking at data in the last 7 days

| summarize count() by bin(TimeGenerated, 1d) //Using Bin to group the data by each day

| render barchart //Looking at the data in a Barchart



Data in Barchart view



Here's a couple additional examples for you to work with. The first one shows IP addresses with the most activity and the second displays disk space available for virtual machines.

And, lastly...if you want something with a really spectacular view, I give you a burst of exciting color.

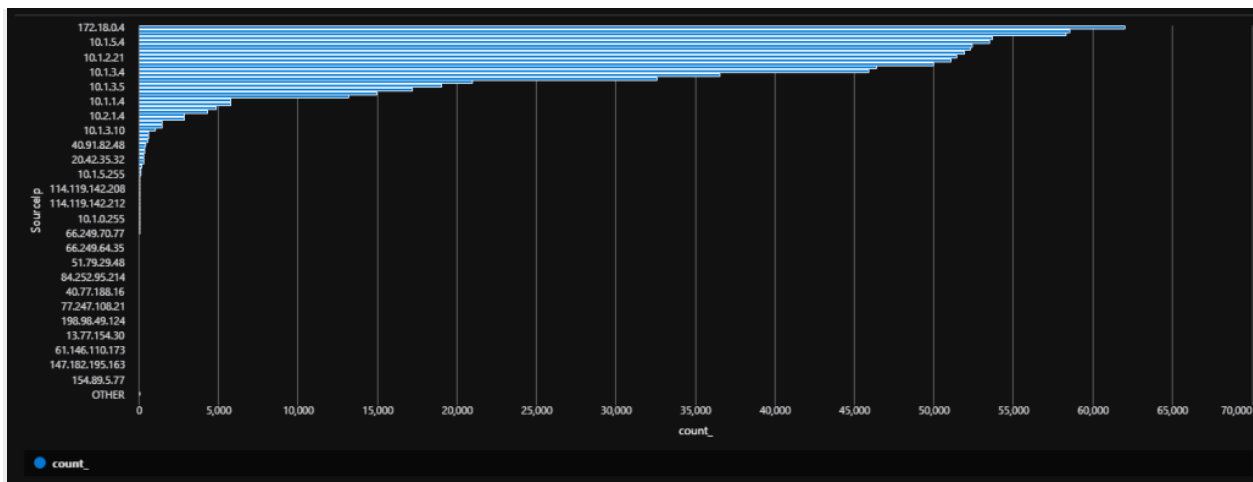
## IP Address Activity

```
VMConnection //Tablename
```

```
| summarize count() by SourceIp //Summarizing found IP addresses
```

```
| sort by count_ desc //Sorting the list in descending order
```

```
| render barchart //Showing the data in a barchart to show activity
```



## Drive Space

Perf //Tablename

| where CounterName == "Free Megabytes" //Looking for free megabytes

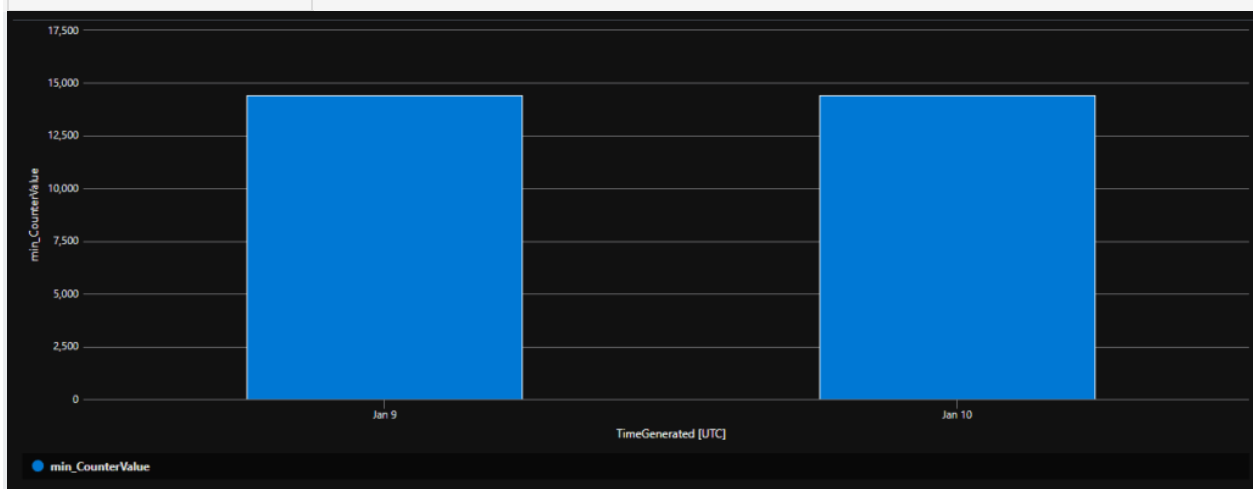
| where InstanceName matches regex "^[A-Z]:\$" //Looking for regular expressions for drive

letters

| summarize min(CounterValue) by bin(TimeGenerated, 1d) //Grouping the data by each drive

letter found

| render columnchart





Ooooo...pretty!

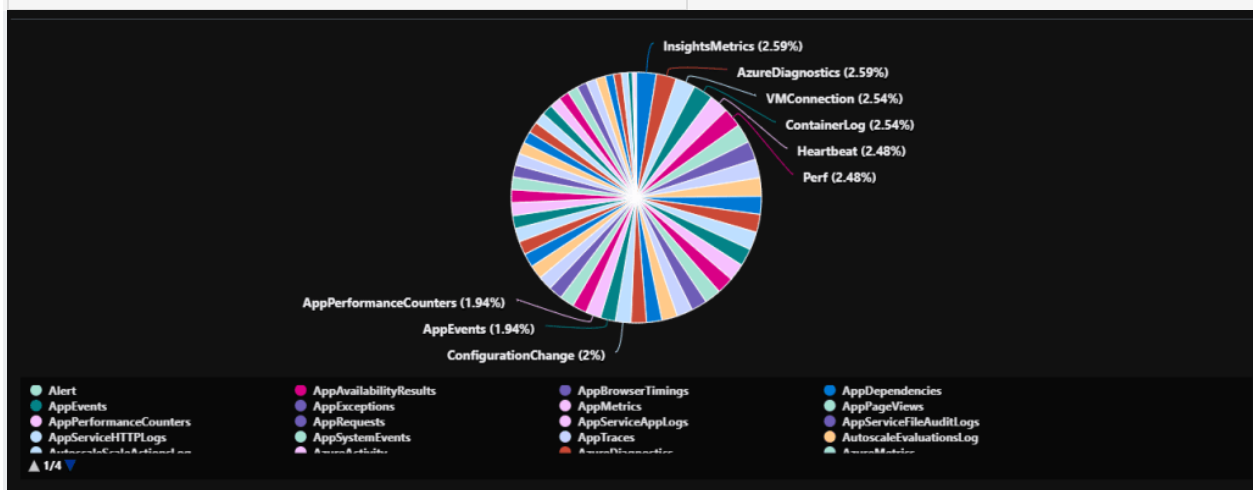
Usage //Tablename

```
| summarize count_per_type=count() by DataType //Creating the numeric value (summary) for types
```

of data

```
| sort by count_per_type desc //Sorted by descending order
```

```
| render piechart //Display the data in a piechart
```



In the next two parts/chapters, we'll get back on track as I start discussing how to manipulate the results that are display using the Extend and Project operators.

## Must Learn KQL Part 13: The Extend Operator

I think it's necessary at this point to do a slight self-recap because in the next few parts/chapters for this Must Learn KQL series ([parts 13-16](#)), I'll talk about how to manipulate the KQL query results so that they can be customized to show exactly what is important to your operations.

Jumping all the way back to [Must Learn KQL Part 3: Workflow](#), realize where we are in the workflow. We're actually very near the end of our original goal. We have discussed finding and organizing data, now it's time to learn how to provide custom views of the data.

Custom data views are important in that each environment is different and each environment's requirements for security will differ – sometimes greatly. Whether its geographical, business political, or something else, the data that is exposed will alter the perception of the organization's risk. So, it's important to expose the right data.

One valuable operator provided with KQL to customize the data views is the *Extend operator*. The Extend operator allows us to build custom columns in real-time in the query results. It allows you to create calculated columns and append them to the results. Understand, though, that we're not creating columns of data that are stored back into the data table, but only generating a column of custom data based on our current request.

Here's a good example...

The following query looks through the *Computer* data column in the *SecurityEvent* table, calculates the character length of the name of each computer found, and produces custom column called 'ComputerNameLength' in the results. Feel free to use this query our the KQL Playground demo environment (<https://aka.ms/LADemo>).

```
SecurityEvent //the table
```

```
| extend ComputerNameLength = strlen(Computer) //creates a new column called ComputerNameLength
```

of the calculation of the number of characters of the computer name in the Computer column

Here's what this will look like...

Run Time range: Last 24 hours Save Share + New alert rule Export Pin to Format query

```
1 SecurityEvent //the table
2 | extend ComputerNameLength = strlen(Computer) //creates a new column called ComputerNameLength of the calculation of the number
3   of characters of the computer name in the Computer column
```

Results Chart Columns Add bookmark Display time (UTC-05:00) Group columns

Completed. Showing partial results from the last 24 hours. 00:12.0 30,000+ records

Showing the first 30,000 results. [Learn more](#) on how to narrow down the result set.

	TimeGenerated [Local Time]	ComputerNameLength	Account	AccountType	Computer	EventSourceName
>	1/18/2022, 5:21:22.667 AM	15	NT AUTHORITY\SYSTEM	User	Windows365Senti	Microsoft Wi
>	1/18/2022, 5:21:29.807 AM	15	NT AUTHORITY\SYSTEM	User	Windows365Senti	Microsoft Wi
>	1/18/2022, 5:22:03.120 AM	15	NT AUTHORITY\SYSTEM	User	Windows365Senti	Microsoft Wi
>	1/18/2022, 5:22:50.603 AM	15			CPC-rodrent-E2	Microsoft Wi
>	1/18/2022, 5:22:50.603 AM	15			CPC-rodrent-E2	Microsoft Wi
>	1/18/2022, 5:22:55.007 AM	15	NT AUTHORITY\SYSTEM	User	CPC-rodrent-E2	Microsoft Wi

Custom column

Again, this column of data is generated in real-time. Once the results have been cleared, this data no longer exists.

## Extend operator syntax

TableName

```
| extend [ColumnName | (ColumnName[, ...]) =] Expression [, ...]
```

In simpler terms, just as before with our standard query workflow we (1) give the query engine the table we want to use, then (2) use the *extend* operator to assign a custom name to a new column, and then (3) insert data into it.

So, using the previous example, I:

1. Designated the *SecurityEvent* table
2. Assigned the name *ComputerNameLength* to the new column
3. Inserted the data I wanted to see. In this case, the hostname length for each computer found in the data.

The data that is inserted into the custom column(s) can be text, number values, calculations, etc., etc., etc. I can use and combine existing table data, or I can fabricate data to be included in the custom column.

In the following example, I'm literally just making stuff up in that the first column called *My\_Calculation*, is just the result of  $8 \times 8$  (64), and *My\_Fabricated\_Data* is just something I wanted to say.

SecurityEvent

| extend My\_Calculation = 8\*8

| extend My\_Fabricated\_Data = "Yay for me!"

The results look like the following...

Feedback Queries Query Explorer

Run Time range : Last 24 hours Save Share + New alert rule Export Pin to Format query

```

1 SecurityEvent
2 | extend My_Calculation = 8*8
3 | extend My_Fabricated_Data = "Yay for me!"
4
5

```

Results Chart Columns Add bookmark Display time (UTC-05:00) Group columns

Completed. Showing partial results from the last 24 hours. 00:13.5 30,000+ rec

Showing the first 30,000 results. [Learn more](#) on how to narrow down the result set.

	TimeGenerated [Local Time]	My_Calculation	My_Fabricated_Data	Account	AccountType
>	1/18/2022, 7:39:55.017 AM	64	Yay for me!	NT AUTHORITY\SYSTEM	User
>	1/18/2022, 7:39:55.030 AM	64	Yay for me!	NT AUTHORITY\SYSTEM	User
>	1/18/2022, 7:39:55.007 AM	64	Yay for me!	WORKGROUP\CPC-RODTRENT...	Machine
>	1/18/2022, 7:39:55.023 AM	64	Yay for me!	WORKGROUP\CPC-RODTRENT...	Machine
>	1/18/2022, 7:42:00.730 AM	64	Yay for me!	NORTHAMERICA\MININT-Q64...	Machine
>	1/18/2022, 7:42:01.113 AM	64	Yay for me!	NORTHAMERICA\MININT-Q64...	Machine
>	1/18/2022, 7:42:04.403 AM	64	Yay for me!		
>	1/18/2022, 7:42:04.590 AM	64	Yay for me!	NORTHAMERICA\rotrent	User
>	1/18/2022, 7:42:07.740 AM	64	Yay for me!	NORTHAMERICA\rotrent	User
>	1/18/2022, 7:42:14.183 AM	64	Yay for me!	NORTHAMERICA\MININT-Q64...	Machine

Just random stuff

But the true beauty for this function is to take existing data, combine it, and display it in meaningful ways.

Take the following as an example of using existing data to make to display it in better ways. Use the following query in the KQL Playground (<https://aka.ms/LADemo>).

```
Perf //table name
```

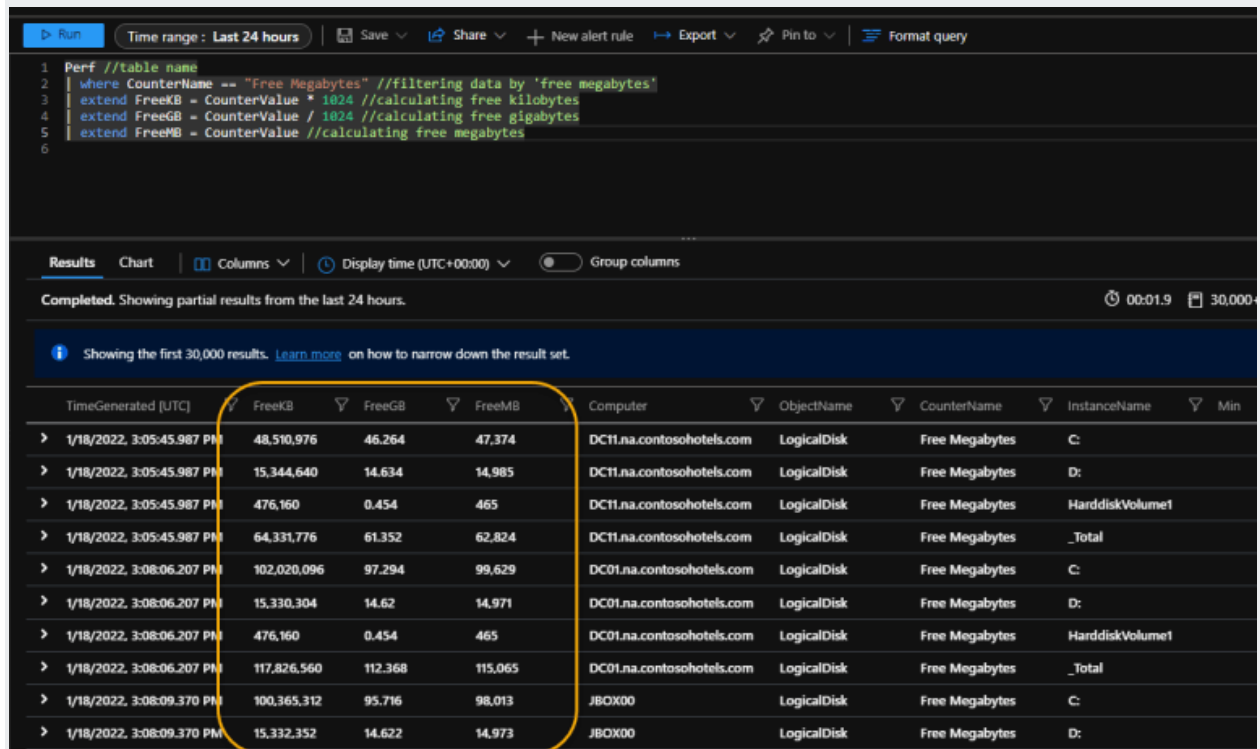
```
| where CounterName == "Free Megabytes" //filtering data by 'free megabytes'
```

```
| extend FreeKB = CounterValue * 1024 //calculating free kilobytes
```

```
| extend FreeGB = CounterValue / 1024 //calculating free gigabytes
```

```
| extend FreeMB = CounterValue //calculating free megabytes
```

This example looks at the *Perf* table to find free disk space on the recorded systems and display it in kilobytes, megabytes, and gigabytes. Your results will look something like the following:



```
1 Perf //table name
2 where CounterName == "Free Megabytes" //filtering data by 'free megabytes'
3 extend FreeKB = CounterValue * 1024 //calculating free kilobytes
4 extend FreeGB = CounterValue / 1024 //calculating free gigabytes
5 extend FreeMB = CounterValue //calculating free megabytes
6
```

Results | Chart | Columns | Display time (UTC+00:00) | Group columns

Completed. Showing partial results from the last 24 hours. 00:01.9 30,000+

Showing the first 30,000 results. [Learn more](#) on how to narrow down the result set.

TimeGenerated [UTC]	FreeKB	FreeGB	FreeMB	Computer	ObjectName	CounterName	InstanceName	Min
1/18/2022, 3:05:45.987 PM	48,510,976	46.264	47.374	DC11.na.contoso-hotels.com	LogicalDisk	Free Megabytes	C:	
1/18/2022, 3:05:45.987 PM	15,344,640	14.634	14.985	DC11.na.contoso-hotels.com	LogicalDisk	Free Megabytes	D:	
1/18/2022, 3:05:45.987 PM	476,160	0.454	465	DC11.na.contoso-hotels.com	LogicalDisk	Free Megabytes	HarddiskVolume1	
1/18/2022, 3:05:45.987 PM	64,331,776	61.352	62.824	DC11.na.contoso-hotels.com	LogicalDisk	Free Megabytes	_Total	
1/18/2022, 3:08:06.207 PM	102,020,096	97.294	99.629	DC01.na.contoso-hotels.com	LogicalDisk	Free Megabytes	C:	
1/18/2022, 3:08:06.207 PM	15,330,304	14.62	14.971	DC01.na.contoso-hotels.com	LogicalDisk	Free Megabytes	D:	
1/18/2022, 3:08:06.207 PM	476,160	0.454	465	DC01.na.contoso-hotels.com	LogicalDisk	Free Megabytes	HarddiskVolume1	
1/18/2022, 3:08:06.207 PM	117,826,560	112.368	115.065	DC01.na.contoso-hotels.com	LogicalDisk	Free Megabytes	_Total	
1/18/2022, 3:08:09.370 PM	100,365,312	95.716	98.013	JBOX00	LogicalDisk	Free Megabytes	C:	
1/18/2022, 3:08:09.370 PM	15,332,352	14.622	14.973	JBOX00	LogicalDisk	Free Megabytes	D:	

## Free disk space

The *Extend* operator is a valuable tool to enable customizing the data that is displayed. As noted, we'll be working with several KQL operators to help develop our own custom views in the next few parts/chapters. But the *Extend* operator is a key creation key tool that you'll find used throughout tools like Microsoft Sentinel to provide things like data parsing and creating custom entities. If you're working with data from custom log files, for example, that data is probably not normalized, and you'll need to expose things like usernames and hostnames that can't be exposed on their own. This is where the *Extend* operator provides huge value. And, as we continue our march to building your first Microsoft Sentinel Analytics Rule, the *Extend* operator is yet another logical step in that process. *Extend* is use quite a bit in Analytics Rules, so understanding it's power and capability is important.

## Must Learn KQL Part 14: The Project Operator

As noted in [part/chapter 13](#) of this series, the next few parts/chapters ([parts 13-16](#)) will be all about how to manipulate the results of the KQL queries. As shown in [part/chapter 13](#), the *Extend operator* allows us to create (and even fabricate) special data to show in the results. On its own, that's hugely valuable. But, also noted throughout this series, the results of the query are the most important part of the process because the types, formats, and ways the data is displayed will allow us to focus on the actual security prospects. And when it comes to identifying threats quickly, efficiency is key.

While [part/chapter 13](#) provided a way to build custom views of the data, that data was still populated among all the rest of the data. Now we get to do something with the data. We get to choose exactly what is displayed to afford our security teams the chance to catch things quickly. We can choose to display our custom data, but then handpick everything else.

This is where the *Project operator* comes into play. Using the *Project operator*, I can tell the query engine the exact data columns to show. In this case, by the way, *Project* is pronounced like as in projector.

The Project operator takes on the following syntax:

```
Tablename  
| project column1, column2, column3
```

Let's take the query we used in [part/chapter 13](#) and add one single line to end. Use the KQL Playground (<https://aka.ms/LADemo>) for your hands-on experience with the following query.

```
Perf //table name  
| where CounterName == "Free Megabytes" //filtering data by 'free megabytes'  
| extend FreeKB = CounterValue * 1024 //calculating free kilobytes  
| extend FreeGB = CounterValue / 1024 //calculating free gigabytes  
| extend FreeMB = CounterValue //calculating free megabytes  
| project Computer, CounterName, FreeGB, FreeMB, FreeKB //only show these columns
```

In this query example, that one single line addition simply tells the query engine to only display the existing *Computer* and *CounterName* columns along with my custom created columns (FreeGB, FreeMB, FreeKB). See that?

Look at the differences in the following comparison images. The top one comes from [part/chapter 13](#) and shows literally every column in the Perf table with my custom created data weaved in. The bottom one is much more concise, precise, and oh so nice.



Time range: Last 24 hours

```

1 Perf //table name
2 where CounterName == "Free Megabytes" //filtering data by 'free megabytes'
3 extend FreeKB = CounterValue * 1024 //calculating free kilobytes
4 extend FreeGB = CounterValue / 1024 //calculating free gigabytes
5 extend FreeMB = CounterValue //calculating free megabytes
6

```

Results Chart Columns Display time (UTC+00:00) Group columns

Completed. Showing partial results from the last 24 hours. 00:02.0 30,000+ records

Showing the first 30,000 results. [Learn more](#) on how to narrow down the result set.

TimeGenerated [UTC]	FreeKB	FreeGB	FreeMB	Computer	ObjectName	CounterName	InstanceName	Min	Max
> 1/20/2022, 5:20:34.817 PM	120,927,232	115.325	118,093	AppFE0000CNF	LogicalDisk	Free Megabytes	C:		
> 1/20/2022, 5:20:34.817 PM	14,748,672	14.065	14,403	AppFE0000CNF	LogicalDisk	Free Megabytes	D:		
> 1/20/2022, 5:20:34.817 PM	476,160	0.454	465	AppFE0000CNF	LogicalDisk	Free Megabytes	HarddiskVolume1		
> 1/20/2022, 5:20:34.817 PM	34,816	0.033	34	AppFE0000CNF	LogicalDisk	Free Megabytes	HarddiskVolume4		
> 1/20/2022, 5:20:34.817 PM	136,186,880	129.878	132,995	AppFE0000CNF	LogicalDisk	Free Megabytes	_Total		
> 1/20/2022, 5:19:24.953 PM	121,100,288	115.49	118,262	AppFE0000CNE	LogicalDisk	Free Megabytes	C:		
> 1/20/2022, 5:19:24.953 PM	14,748,672	14.065	14,403	AppFE0000CNE	LogicalDisk	Free Megabytes	D:		
> 1/20/2022, 5:19:24.953 PM	476,160	0.454	465	AppFE0000CNE	LogicalDisk	Free Megabytes	HarddiskVolume1		
> 1/20/2022, 5:19:24.953 PM	34,816	0.033	34	AppFE0000CNE	LogicalDisk	Free Megabytes	HarddiskVolume4		
> 1/20/2022, 5:19:24.960 PM	136,359,936	130.043	133,164	AppFE0000CNE	LogicalDisk	Free Megabytes	_Total		

Time range: Last 24 hours

```

1 Perf //table name
2 where CounterName == "Free Megabytes" //filtering data by 'free megabytes'
3 extend FreeKB = CounterValue * 1024 //calculating free kilobytes
4 extend FreeGB = CounterValue / 1024 //calculating free gigabytes
5 extend FreeMB = CounterValue //calculating free megabytes
6 project Computer, CounterName, FreeGB, FreeMB, FreeKB //only show these columns
7

```

Results Chart Columns Display time (UTC+00:00) Group columns

Completed. Showing partial results from the last 24 hours. 00:01.2 30,000+ records

Showing the first 30,000 results. [Learn more](#) on how to narrow down the result set.

Computer	CounterName	FreeGB	FreeMB	FreeKB
> JBOX10	Free Megabytes	95.402	97.692	100,036,608
> JBOX10	Free Megabytes	14.654	15,006	15,366,144
> JBOX10	Free Megabytes	0.454	465	476,160
> JBOX10	Free Megabytes	110.511	113,163	115,878,912
> AppBE00.na.contosohotels....	Free Megabytes	97.729	100,074	102,475,776
> AppBE00.na.contosohotels....	Free Megabytes	14.611	14,962	15,321,088
> AppBE00.na.contosohotels....	Free Megabytes	0.454	465	476,160
> AppBE00.na.contosohotels....	Free Megabytes	112.794	115,501	118,273,024
> RETAILVM01	Free Megabytes	102.589	105,051	107,572,224
> RETAILVM01	Free Megabytes	14.64	14,991	15,350,784
> RETAILVM01	Free Megabytes	0.454	465	476,160

Page 1 of 600 50 items per page 1 - 50 of 30000 items

The Project operator may seem like a simple tool, but its hugely powerful, giving you data choice.

But there's more to it. As the TV detective, **Columbo**, used to say: just one more thing...

The Project operator has more depth than you might realize.

First off, like the [Extend operator](#), you can use Project to create custom columns. For example, the following query eliminates the [Extend operator](#) completely and just creates the same custom columns as before.

```
Perf //table name
| where CounterName == "Free Megabytes" //filtering data by 'free megabytes'
| project Computer, CounterName, FreeGB=CounterValue / 1024, FreeMB = CounterValue, FreeKB = CounterValue * 1024 //only show these columns
```

You might think based on this that, hey, I'll never need to use the [Extend operator](#) again. But, no, that's not the case and I'll dig deeper into this as we get closer in the series to building an actual Analytics Rule ([see the TOC](#)).



And, then there are also some other *Project* options that are so hugely valuable that they have their own operator reference page and are actually considered their own operators. However, in respect to our discussion in this series you should absolutely keep each of these in your toolbelt and have knowledge about them.

Here's why:

**Project-away** – *Select what columns from the input to exclude from the output. Project-away* is probably one of my favorite *Project* operator variants because it's an efficiency tool. Most tables have 25 or more data columns stored inside. What if you want to display all columns *except* for 4 or 5? Would you use the *Project* operator and just manually type out 20 columns to show? I hope not – once you understand how *Project-away* works. Using *Project-away* you can effectively tell the query engine to display **ALL** columns **EXCEPT** the one's you list in the *Project-away* statement. Notice also that wildcards are supported:

```
Tablename  
| project-away column1, column2, column3*
```

**Project-keep** – *Select what columns from the input to keep in the output.*

This *Project* variant is essentially the default operation. Hint: Just use the standard *Project* option.

**Project-rename** – *Renames columns in the result output.* This option gives you the ability to rename column headers during query-time. Don't like a column name? Or, maybe your team has standardized on Computer as a specific value for the Hostname column. This is how you can change the column name in the results. Note that this does not change the data column name in the table – just in the query-time results.

```
Tablename  
| project-rename my_new_column_name = old_column_name
```

**Project-reorder** – *Reorders columns in the result output.* Most generally, the order of columns in the results will be determined based on their original order in the table. But, alas, sometimes even that doesn't hold true. If you want to make sure to display the columns in a specific order without turning their fate over to chance, use *project-reorder*.

```
Tablename  
| project-reorder column2, column3, column1
```

## Must Learn KQL Part 15: The Distinct Operator

The next couple parts/chapters in the Must Learn KQL series are shorter ones as we complete the *series-within-the-series* ([parts 13-16](#)) on manipulating the query results. So, in this mini-series far we've talked about how to create custom columns with the [Extend operator](#) and shown how to display (or not display) specific data in the results using the [Project operator](#). But what if you need to get even more granular with the data that is presented? This is where the [Distinct operator](#) comes in.

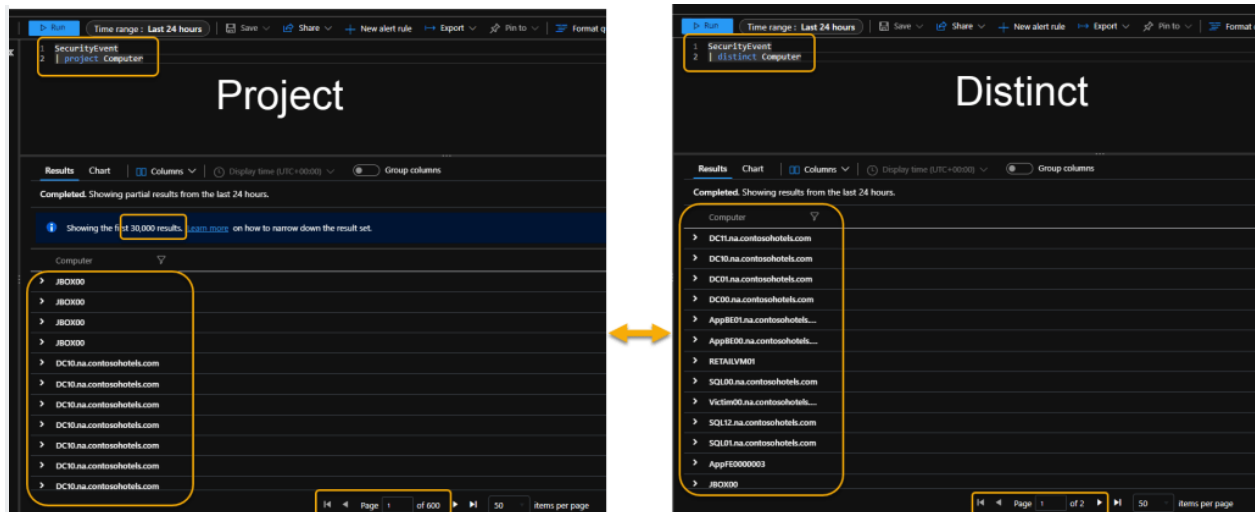
As you can imagine by the operator's name, the [Distinct operator](#) delivers results based on a *distinct* combination of the columns you provide.

For example, if I have 100's of computers stored in the table, each with their own combination of activity and data, but I only want to know each computer name, I would supply a KQL query similar to the following:

```
SecurityEvent //the table
| distinct Computer //show distinct computer names
```

Feel free to use the KQL Playground (<https://aka.ms/LADemo>) we've used through to try this query out yourself.

Look at the results differences between using just the [Project operator](#) against the [Distinct operator](#). Notice the differences in simplicity of *what* is displayed along with the *volume* of what is displayed.



## Project versus Distinct

Distinct can also be used for more than one data column as is shown above and is generally intended (*as I mentioned above*) to produce a combination of the columns you provide. The beauty of the [Distinct operator](#) is that it allows you to get extremely precise in what is returned, which is hugely important when using KQL to perform security Hunting operations – *which I'll cover after we've achieved our goal in this series of creating our first Analytics Rule for Microsoft Sentinel (watch the [TOC](#) for details).*

Before handing you off to a series of hands-on opportunity examples, let's take some of our accumulated knowledge and apply it to this same scenario. What if we wanted to not only show the distinct computer names (*as in the example above*), but also needed to show how much activity each computer has been responsible for over the last 24 hours?

Don't cheat! Think about it for a second. What operators would you combine with *Distinct*? How would you get a *summarized count* of *distinct* computers? **<== there's a hint in there**

Remember, story problems and developing storylines are the basis for all security. That doesn't just mean things like who did it and why, but also includes how to expose the data to show the story. That's important.

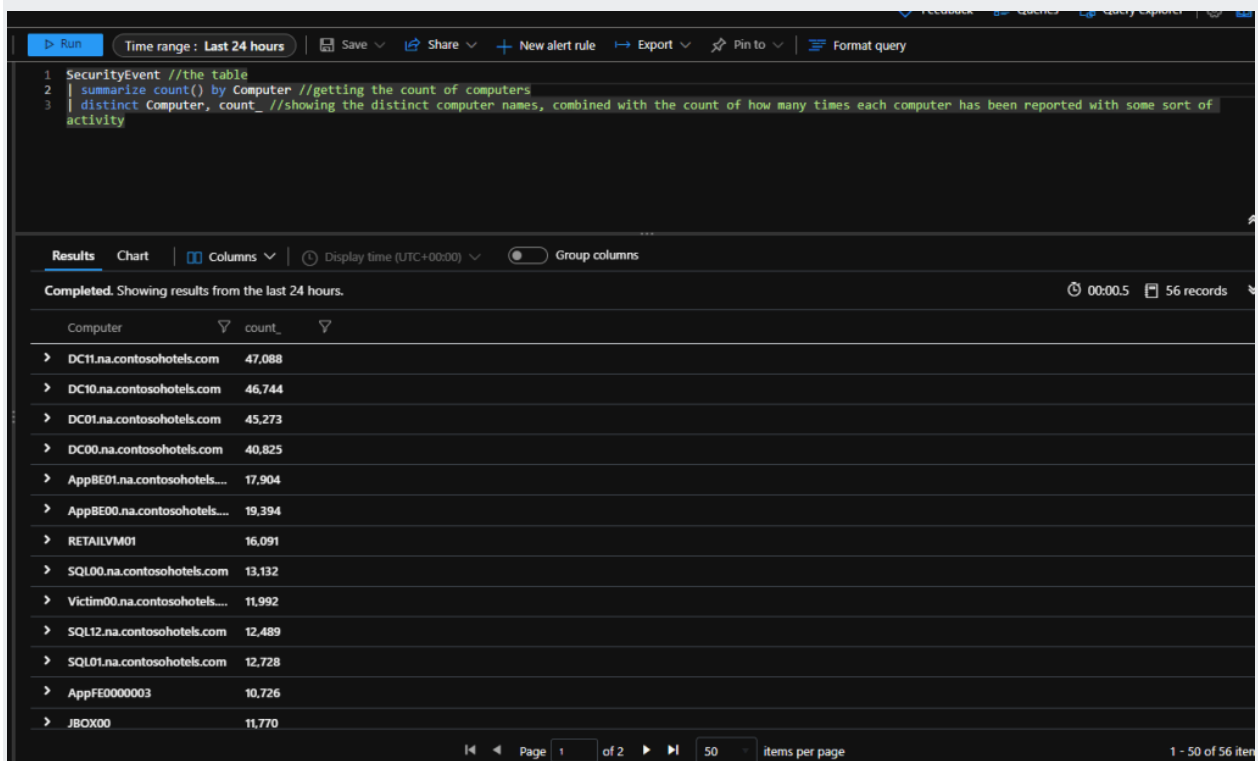
OK...here's who I would supply the result from the ask:

```
SecurityEvent //the table
| summarize count() by Computer //getting the count of computers
```

```
| distinct Computer, count_ //showing the distinct computer names, combined with the count of how many times each computer has been reported with some sort of activity
```

In the example, notice that I've used [summarize](#) to get a [count](#) of all Computers first, then combined the Computer data column with the count in the last line using *Distinct*.

Your results using the KQL Playground (<https://aka.ms/LADemo>) will be something similar to the following...



The screenshot shows the KQL Playground interface. The query is:

```
1 SecurityEvent //the table
2 | summarize count_ by Computer //getting the count of computers
3 | distinct Computer, count_ //showing the distinct computer names, combined with the count of how many times each computer has been reported with some sort of activity
```

The results are displayed in a table with columns 'Computer' and 'count\_'. The table shows 15 rows of data, including computer names and their corresponding counts. The interface also shows a 'Time range' of 'Last 24 hours' and a 'Display time' of 'UTC+00:00'.

Computer	count_
DC11.na.contoso-hotels.com	47,088
DC10.na.contoso-hotels.com	46,744
DC01.na.contoso-hotels.com	45,273
DC00.na.contoso-hotels.com	40,825
AppBE01.na.contoso-hotels.com	17,904
AppBE00.na.contoso-hotels.com	19,394
RETAILVM01	16,091
SQL00.na.contoso-hotels.com	13,132
Victim00.na.contoso-hotels.com	11,992
SQL12.na.contoso-hotels.com	12,489
SQL01.na.contoso-hotels.com	12,728
AppFE0000003	10,726
JBOX00	11,770

Count of Computer Activity

## All YOU, baby!



The following series of KQL queries gives you a chance to get some hands-on experience with more of the Distinct operator and a chance to shine your growing expertise. While the following examples are not specifically security related, I

believe you will find them interesting because KQL transcends one product, one workload, or one area of focus. It's good to reiterate this. KQL is important to anyone working in Azure and will only become more critical as time progresses. You need to elbow nudge your colleagues over this. Your KQL learning in this Must Learn KQL series could lead to a whole new career path.

Take the following examples and run them in the KQL Playground (<https://aka.ms/LADemo>) and then prepare for the last part/chapter of this series-within-a-series on manipulating results.

*P.S. If you're just being introduced to this series, I entreat you to [start at the beginning](#) before digging into the examples – unless of course, you're already a KQL master.*

```
Perf //the table
| distinct Computer //find all the computers that are reporting performance data to Log Analytics
```

```
Perf //the table
| distinct ObjectName //Using the same performance example, finding all the object types that we have performance data for
```

```
Perf //the table
| distinct ObjectName, CounterName //We want to see all the metrics for each object, in this case CounterName
```

```
UpdateSummary //the table
| distinct Computer, WindowsUpdateSetting //We can get our Windows Update Settings for all servers we're managing with the Update Management solution
```

```
UpdateSummary //the table
| distinct Computer, WindowsUpdateSetting, OSVersion, OldestMissingSecurityUpdateInDays //However, we're not limited to just one or two fields. We can add more, in this example we'll get our servers, their update setting, OS version and the oldest update they need in days
```

```
Update //the table
| where UpdateState == "Needed" //retrieve only those systems where updates are needed
| distinct Computer, KBID, Title //Finally, we can quickly build a report of systems needing updates, the KB number and title of the update
```

## Must Learn KQL Part 16: The Order/Sort and Top Operators

In this last part/chapter of the series-within-the-series for data view manipulation, I'm going to combine a couple operator types. Looking at the title of this part/chapter, it may seem that I'm focused on three operators (order, sort, and top), but really – like the [Limit/Take operators from Part 9](#) – Order and Sort provide functionally no difference. This is one of those situations, again, where it becomes personal preference which one to use. In fact, when you read through the KQL reference doc it will tell you that...

*The order operator is an alias to the sort operator.*

...and then tell you to go check out the [Sort operator page](#).

So, let's focus on that first.

The Order By/Sort By operator type enables you to sort data columns in the query results so you can view the data first in a way that's more meaningful. For example, the following query (which you can use in the KQL Playground <https://aka.ms/LADemo>), queries the SecurityEvent table for the last 7 days of data and shows a random 100 records in descending order by the time each returned record was generated.

```
SecurityEvent //the table
| where TimeGenerated > ago(7d) //look at data in the last 7 days
| order by TimeGenerated desc //sort or order the TimeGenerated data column in descending order
| limit 100 //show 100 random records
```

There's a couple important things to call out about the Order By/Sort By operations:

1. You can Sort by multiple columns and each column by different directions. For example, replace the Order By line above with the following: `| order by TimeGenerated desc, Computer asc`
2. The default view returned for data is descending order (*desc*).



3. If you are sorting by a data column that has null values (empty records), those will be displayed first using the default order (*desc*).

You have the option with Order/Sort to directly – as part of the sorting – to adjust where the nulls show up by adding either a ***nulls first*** or ***nulls last*** option as shown in the next example.

```
SecurityEvent //the table
| where TimeGenerated > ago(7d) //look at data in the last 7 days
| order by TimeGenerated desc nulls first //sort or order the TimeGenerated data column in
descending order, showing nulls first
| limit 100 //show 100 random records
```

**TIP:** If the *null records* thing bothers you like it does me (*must be an OCD thing*), you may want to modify your query so that null records aren't returned at all. Here's a simple modification to the above query to stop showing data if the *Account* data column is empty.

```
SecurityEvent //the table
| where TimeGenerated > ago(7d) and isnottnull(Account) //look at data in the last 7 days where the
Account column isn't empty
| order by TimeGenerated desc //sort or order the TimeGenerated data column in descending order
| limit 100 //show 100 random records
```

Just be careful with this. Sometimes, null columns can be an important delimiter.

Lastly, to continue to improve and hone your query knowledge – *particularly for efficiency* – the Top operator can be used to simplify our example. Ascending and Descending order work the same for Top as it does for Order/Sort, but we can combine expressions using Top, as is the case in the following example. Plus, the Top operator is a great way to retrieve the most recent records instead of always relying on random samples.

```
SecurityEvent //the table
| top 100 by TimeGenerated desc //Retrieving the top 100 records sorted in descending order by
TimeGenerated
```

See what I did there? The Top operator is performing, essentially, the same operation as before, but it has simplified the query that it is required. In this case, though, I'm returning top values instead of the random ones that the Limit operator supplies. Top also provides the same options as Order/Sort for null values, so you can choose where to place the empty data columns in the display.

## Must Learn KQL Part 17: The Let Statement

Going way back to [part 3](#) when I talked about the standard workflow, you might remember me saying...

***Even though the structure can deviate, understanding a common workflow of a KQL query can have powerful results and help you develop the logic needed to build your own workflows when it's time to create your own queries.***

**Rod Trent, November 19, 2021**

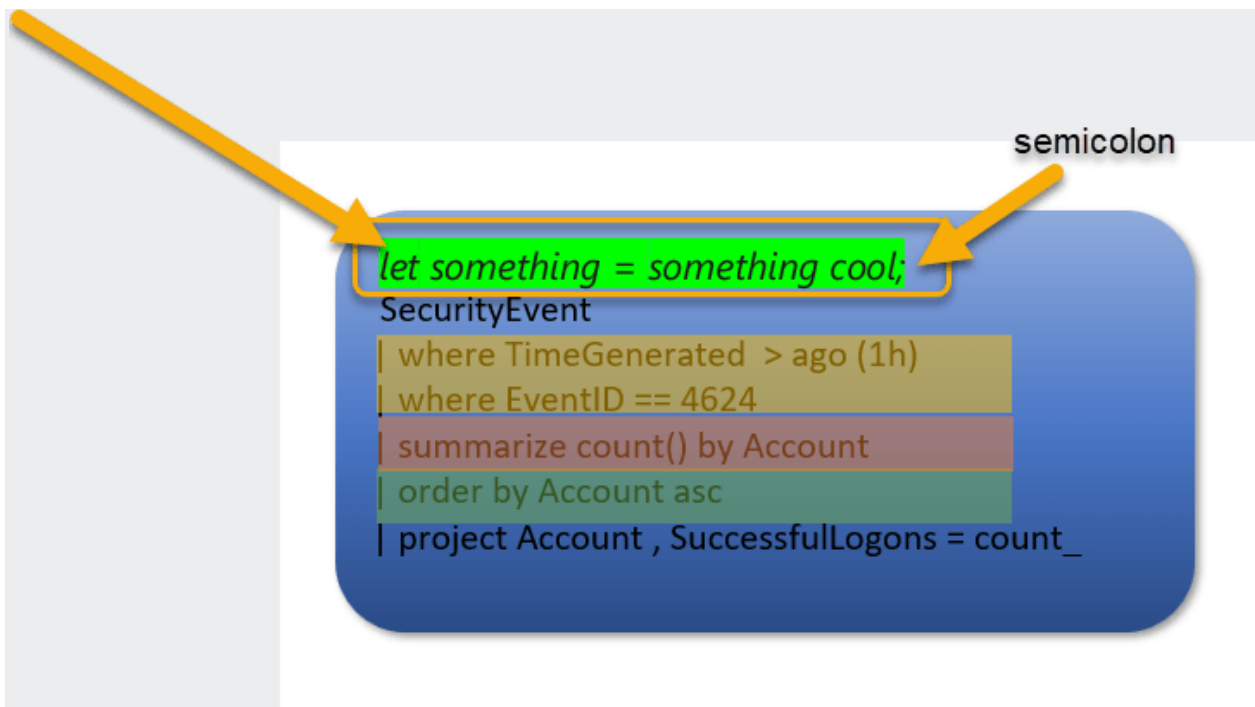
In this part/chapter of the Must learn KQL series, I'm going to focus on one of those deviations. As you'll see, the [Let statement](#) can deviate from the norm because it's generally assumed that it is positioned before the query event begins because of what it does.

So, what does the [Let statement](#) do?

The easiest way to put it is that it simply allows you to create variables. This makes sense to a lot folks who script or program and it's not dissimilar.

These variables are stored in memory during query execution and can be used throughout the rest of the query. It's considered a best practice and is used for developing better performing queries and query code re-use.

Most generally the [Let statement](#) will show up at the beginning of the main query, as shown in the following slight modification of the original workflow we talked about in [Part 3](#).



Normal location of the Let statement

Another important thing to note is that the statement must be finalized. It *is* a statement, after all. The Let statement ends with the semicolon (;) character. This tells the query engine that the variable has been created and needs to be stored before carrying on with the rest of the query.

The best way to understand this, is to just dig into some examples. Please use the KQL Playground (<https://aka.ms/LADemo>) to get hands-on for the following types of Let statements.

## Creating Variables from Scratch

The first method of using the Let statement is simply to generate your own data. In the following example, I've created a *timeOffset* variable that provides a time value of 7 days, I've created another variable called *discardEventID* that sets our Event ID to 4688 which records when a new process on a computer has been spawned.

The *timeOffset* is used to create a time range of between 7 and 14 days in which to look at data. The *discardEventID* is used to show everything BUT 4688 in the results.

```
let timeOffset = 7d; //Setting the offset variable
let discardEventId = 4688; //assigning new process as the event ID
SecurityEvent //the table
```

```
| where TimeGenerated > ago(timeOffset*2) and TimeGenerated < ago(timeOffset) //Setting a specific  
time range of between 7 and 14 days  
| where EventID != discardEventId //showing all events but 4688
```

## Creating Variables from Existing Data

Another type of [Let statement](#) is one that pulls data from existing tables. Essentially, you turn the results of a query into a variable. The following example assigns one query to the login variable. The second query assigns results to the logout variable. And, then to wrap up the full results, the query then merges (joins – which I’ll cover just a bit later in this series) both variables (login and logout) to show login and logout times for all accounts.

```
let login = SecurityEvent //Setting the login variable based on a full query  
| where TimeGenerated > ago(1h) //look at records in the last hour  
| where EventID == '4624' //setting the event ID to successful login  
| project Account, TargetLogonId, loginTime = TimeGenerated; //creating the full output, notice the  
semicolon to end the let statement  
let logout = SecurityEvent //Setting the logout variable based on a full query  
| where TimeGenerated > ago(1h) //look at records in the last hour  
| where EventID == '4634' //setting the event ID to successful logoff  
| project Account, TargetLogonId, logoutTime = TimeGenerated; //creating the full output, notice the  
semicolon to end the let statement  
login //Accessing the login output  
| join kind=leftouter logout on TargetLogonId //joining login output with logout output  
| project Account, loginTime, logoutTime //Showing login and logout times for each account
```

## Creating Variables from Microsoft Sentinel Watchlists

And, finally, Microsoft Sentinel customers should know that using the [Let statement](#) enables them to use the Watchlist feature with their Analytics Rules.

Now, I apologize for this, but the following examples cannot be used with the KQL Playground (<https://aka.ms/LADemo>) because the KQL Playground is not enabled for Microsoft Sentinel. As a Microsoft Sentinel customer, you can use these in your own Sentinel environment. However, notice that I have a Watchlist called *FeodoTracker* – you probably don’t. Also, my *FeodoTracker* Watchlist has a data column called *DstIP* (destination IP address) – you probably don’t.

However, I wanted to include these examples for those working with Watchlists. These examples represent Watchlist basics. The [Let statement](#) is used to build a variable for data that exists in the Watchlist. In the first example, I’m looking for IPs

that exist (**in**) in the Watchlist. In the second one, I'm looking for IPs that don't (**!in**) exist in the Watchlist.

And, while not part of our [Let statement](#) topic, the last two examples show how to call Watchlist data in the midst of the query instead of assigning variable. Best practice is to use the [Let statement](#), but I've supplied these examples to show that it's possible.

```
//Watchlist as a variable, where the requested data is in the list
let watchlist = (_GetWatchlist('FeodoTracker') | project DstIP);
Heartbeat
| where ComputerIP in (watchlist)

//Watchlist as a variable, where the requested data is not in the list
let watchlist = (_GetWatchlist('FeodoTracker') | project DstIP);
Heartbeat
| where ComputerIP !in (watchlist)

//Watchlist inline with the query, where the requested data is in the list
Heartbeat
| where ComputerIP in (
    (_GetWatchlist('FeodoTracker')
    | project DstIP)
)

//Watchlist inline with the query, where the requested data is not in the list
Heartbeat
| where ComputerIP !in (
    (_GetWatchlist('FeodoTracker')
    | project DstIP)
)
```

Not familiar with Microsoft Sentinel Watchlists or what to do with your Let statement for Watchlists after you create it? See: [Use watchlists in Microsoft Sentinel](#)

## Must Learn KQL Part 18: The Union Operator

As I did with parts/chapters [13-16](#) of this series for the *series-within-the-series* for data view manipulation, this part/chapter and the next form another mini-series of sorts. The [Union](#) and [Join](#) operators are important parts of the KQL journey as they represent opportunities to combine data from tables in different ways.

Before jumping directly off into talking about the Union operator, I think it's best to start with describing the differences between Union and Join. Knowing the differences will allow you to determine which one to use for which scenario.

*Union* allows you to take the data from two or more tables and display the results (all rows from all tables) together. *Join*, on the other hand, is intended to produce more specific results by joining rows of just two tables through matching the values of columns you specify. You'll see the differences once we get through this mini-series and you can get hands-on with the examples. I highly suggest taking the examples from this part/chapter and running them against the examples of Part 19 on the Join operator to get a proper comparison.

There's a lot to the Union operator, so I suggest reviewing the [reference page](#) for all additional options, including things like *kind=inner(common columns)*, *outer (all columns- default)*, and *isfuzzy*. I'll discuss Union more in the Advanced series, [Addicted to KQL](#), but for our purposes for the Must Learn KQL journey what's important to know are the following:

- Union supports wildcard to union multiple tables (union Security\*)
- Union can be used to merge tables from different Log Analytics Workspaces (or clusters)

For most of your operations in the Microsoft security tools like Microsoft Sentinel for creating Analytics Rules (covered in [Part 20](#), the last part/chapter of the *Must Learn KQL series*), you'll make use of the [Join operator](#) because of its ability to hone directly into specific results. Union, though, is an important tool for hunting in Microsoft Sentinel and Advanced Hunting in Defender.

To get started with the Union operator, use the following examples in the KQL Playground (<https://aka.ms/LADemo>).

This following query merges the SecurityEvent and Heartbeat tables and then displays each hostname (computer) stored in both tables and how many times each computer is recorded for some sort of activity.

```
SecurityEvent //the table
| union Heartbeat //merging SecurityEvent table with the Heartbeat table
| summarize count() by Computer //showing all computers from both tables and how many times
```

This next query example is the same as before but merging an additional table (*SecurityAlert*) to show the data from three tables instead of two.

```
SecurityEvent //the table
| union Heartbeat, SecurityAlert //merging SecurityEvent table with the Heartbeat table and
SecurityAlert
| summarize count() by Computer //showing all computers from all tables and how many times they are
referenced
```

The following example introduces a couple changes to the first two queries in that it merges all tables that start with 'Sec' (*notice the wildcard character*) and sorts the computers in alphabetical ascending order (the last line).

```
SecurityEvent //the table
| union Sec* //merging together all tables beginning with 'Sec'
| summarize count() by Computer //showing all computers from all tables and how many times they are
referenced
| sort by Computer asc //displaying Computer names in ascending order
```



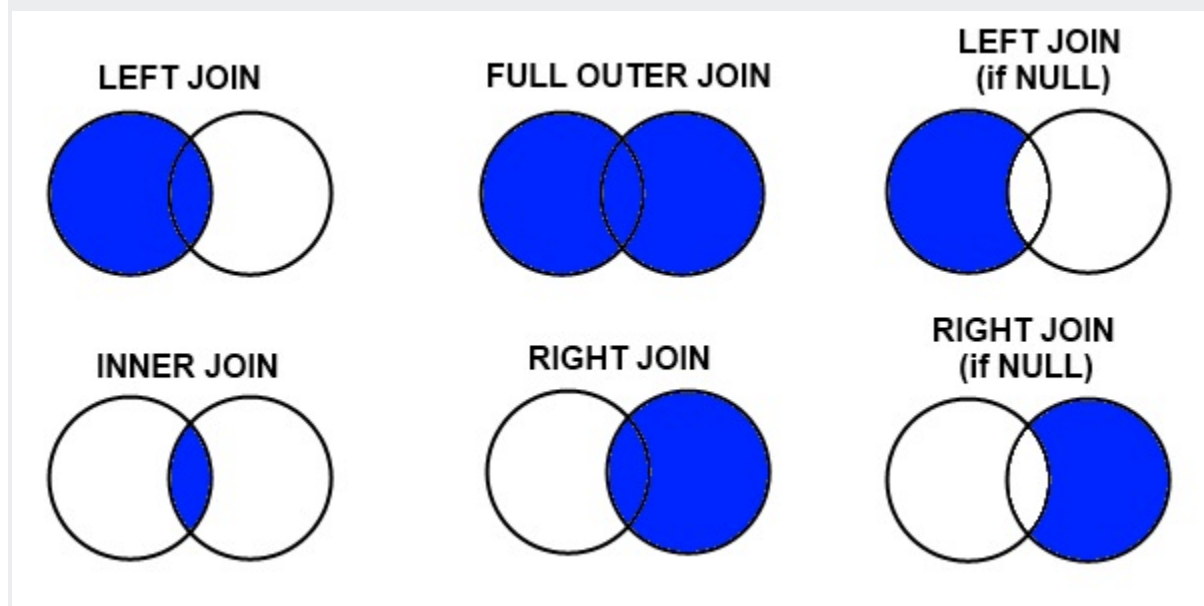
## Must Learn KQL Part 19: The Join Operator

As noted in [part/chapter 18](#), this mini-series on merging data contains two different principles. Reiterated from the last part/chapter...

**Union** allows you to take the data from two or more tables and display the results (all rows from all tables) together. **Join**, on the other hand, is intended to produce more specific results by joining rows of just two tables through matching the values of columns you specify.

There's quite a bit more to the [Join operator](#) (and Join, in general) than I'll cover in this part/chapter. I want to make sure to keep this focused on those things necessary to help build your first Microsoft Sentinel Analytics Rule in the final part/chapter of this series.

Join, merges the rows of two tables (left table and right table) to form a new pseudo-table by matching values of the specified column(s) from each table. Just like any other query language's Join, the KQL Join operator supports the following Join methods along with some additional nuanced options – *with inner Join being the default*.



Joining tables and data

The syntax for the Join operator is as follows:

*LeftTable* |join [*JoinParameters*] (*RightTable*) on *Attributes*

Use the following example in the KQL Playground (<https://aka.ms/LADemo>). This example joins together the *SecurityEvent* and *Heartbeat* tables on the common *Computer* column. It then filters all Computers by the 4688 Event ID (newly spawned process) and shows the Computer name and the installed OS and versioning.

```
SecurityEvent //table name
| join Heartbeat on Computer //joining SecurityEvent with Heartbeat on the common Computer column
| where EventID == "4688" //Looking for Event ID for new process
| project Computer, OSType, OSMajorVersion, Version //Displaying data from both tables
```

Your results should look similar to the following:

The screenshot shows the KQL Playground interface. At the top, there's a toolbar with buttons for 'Run', 'Time range: Last 24 hours', 'Save', 'Share', 'New alert rule', 'Export', 'Pin to', and 'Format query'. Below the toolbar is a query editor with the following code:

```
1 SecurityEvent //table name
2 | join Heartbeat on Computer //joining SecurityEvent with Heartbeat on the common Computer column
3 | where EventID == "4688" //Looking for Event ID for new process
4 | project Computer, OSType, OSMajorVersion, Version //Displaying data from both tables
```

Below the query editor, there's a 'Results' tab selected. It shows a table with the following columns: Computer, OSType, OSMajorVersion, and Version. The table contains 12 rows of data, all with 'Windows' as the OSType and '10' as the OSMajorVersion. The Version column shows various values like '10.20.18064.0', '1.0.12.0', and '1.1.2.0'.

Computer	OSType	OSMajorVersion	Version
JBOX00	Windows	10	10.20.18064.0
DC00.na.contosohotels.com	Windows	10	1.0.12.0
DC01.na.contosohotels.com	Windows	10	10.20.18064.0
DC01.na.contosohotels.com	Windows	10	1.0.12.0
SQL12.na.contosohotels.com	Windows	10	10.20.18064.0
DC00.na.contosohotels.com	Windows	10	10.20.18064.0
DC11.na.contosohotels.com	Windows	10	10.20.18053.0
DC01.na.contosohotels.com	Windows	10	1.0.12.0
JBOX00	Windows	10	1.0.12.0
JBOX10	Windows	10	10.20.18053.0
DC11.na.contosohotels.com	Windows	10	1.1.2.0

Results of Joining by the Computer Column



Here's something fun. To change the *kind* (or, *flavor*) of Join, you simply add a *kind* option like so.

| join **kind=inner** Heartbeat on Computer

Changing the flavor of join will alter how and what data is displayed. Changing our original Join query example with the inner flavor or join will produce results like the following (*note the results display difference from before*)....

The screenshot shows the KQL Playground interface. At the top, there's a toolbar with buttons for 'Run', 'Time range: Last 24 hours', 'Save', 'Share', 'New alert rule', 'Export', 'Pin to', and 'Format query'. Below the toolbar, a query is entered in a text area:

```
1 SecurityEvent //table name
2 | join kind=inner Heartbeat on Computer //joining SecurityEvent with Heartbeat on the common Computer column
3 | where EventID == "4688" //Looking for Event ID for new process
4 | project Computer, OSType, OSMajorVersion, Version //Displaying data from both tables
```

Below the query editor, the 'Results' tab is selected. It shows a table with the following columns: Computer, OSType, OSMajorVersion, and Version. The table contains 8 rows of data, all with the same values: RETAILVM01, Windows, 10, and 10.20.18053.0.

Computer	OSType	OSMajorVersion	Version
RETAILVM01	Windows	10	10.20.18053.0
RETAILVM01	Windows	10	10.20.18053.0
RETAILVM01	Windows	10	10.20.18053.0
RETAILVM01	Windows	10	10.20.18053.0
RETAILVM01	Windows	10	10.20.18053.0
RETAILVM01	Windows	10	10.20.18053.0
RETAILVM01	Windows	10	10.20.18053.0
RETAILVM01	Windows	10	10.20.18053.0

Try the following on your own in the KQL Playground (<https://aka.ms/LADemo>):

- | join kind=innerunique Heartbeat on Computer
- | join kind=leftouter Heartbeat on Computer

- | *join kind=rightouter Heartbeat on Computer*
- | *join kind=fullouter Heartbeat on Computer*

In the advanced series, [Addicted to KQL](#), I'll dig deeper into the other use cases for Join. If you're champing at the bit to learn more now and happen to be a Star Wars nut, check out Jing's KQL Tutorial on the Join operator on YouTube: [KQL Tutorial Series | Joining Tables \(Demo\) | EP5](#)

## Must Learn KQL Part 20: Building Your First Microsoft Sentinel Analytics Rule

The intent of this series has been to enable you to understand the structure, flow, capability, and simplicity of the KQL query language. Way back in [part/chapter 3](#), I said...

*I tell customers all the time that it's not necessary to be a pro at creating KQL queries. It's OK not to be a pro on day 1 and still be able to use tools like Microsoft Sentinel to monitor security for the environment. As long as you understand the workflow of the query and can comprehend it line-by-line, you'll be fine. Because ultimately, the query is unimportant. Seriously. What's important for our efforts as security folks is the results of the query. The results contain the critical information we need to understand if a threat exists and then – if it does exist – how that threat occurred from compromise to impact.*

And that remains the case. I'll dig much, much deeper into KQL in the [Addicted to KQL series](#), but for our purposes here in the *Must Learn KQL series*, you should have become comfortable with eyeing a query and understanding it's intent line-by-line. If you're just joining us because this part/chapter has the words *Microsoft Sentinel* and *Analytics Rules* on it, you're starting at the wrong spot. I entreat you to jump back to the beginning and ingest this series in the methodical, logical manner it was intended.

*(If you have suggestions for the TOC for the Addicted to KQL series, [let me know](#). Current TOC is here: <https://aka.ms/Addicted2KQL>)*

Keeping with the original plan to build your first Analytics Rule, we're going to work together to understand an existing Analytics Rule example that you can use in your own Microsoft Sentinel environment. This example takes many of the concepts and operators we've learned together on this journey, so you should be intimately familiar with them. And, if all works well, you should be on your way to mastering KQL and hungry for what's next. If you're like me, this stuff just geeks you out. You may find yourself thinking about Joins and Summarizations at strange times, but

don't fret – you are not alone. KQL can do that. I regularly zone off thinking about table schema while the wife is telling me something that's probably important. Heck even my email signature is homage to KQL:

```
Rod Trent
| where Title == 'SENIOR CLOUD SECURITY ADVOCATE'
  and Focus == 'Microsoft Sentinel/Defender for Everything/Cybersecurity'
  and Group == 'C+E DevRel CA'
| project
Office: 513-826-9255,
Email: rod.trent@microsoft.com,
Twitter: @rodtrent,
Blog: aka.ms/RodBlog,
LinkedIn: Rod Trent | LinkedIn
Sentinel News: aka.ms/MicrosoftSentinelNewsletter
Defender News: aka.ms/MicrosoftDefenderNewsletter
```

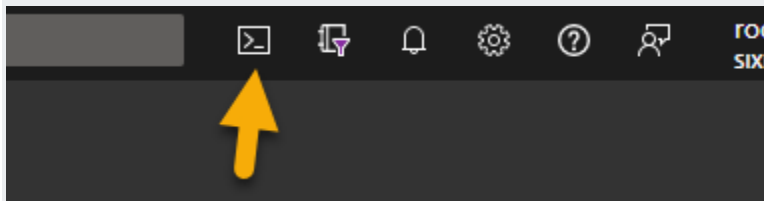
My email signature –

make it your own!

Feel free to steal that, by the way.

## Analytics Rule

So, let's take a well-established Analytics Rule and pick it apart. This one is intended to capture individuals that run Cloud Shell in the Azure portal. This query/rule needs to be run in your own Microsoft Sentinel environment and not in the KQL Playground (<https://aka.ms/LADemo>). To get results, someone must have run Cloud Shell in recent record.



Run Cloud Shell

Cloud Shell activity is logged in the AzureActivity table. So, our first line of the query will be:

```
AzureActivity //the table - this is where Cloud Shell activity is logged
```

Once we've identified the table we need to query against, as per the [Workflow discussion in part/chapter 3](#), it's time to start filtering the data. So, using the [Where](#)

[operator covered in part/chapter 8](#), let's dig into what exactly identifies Cloud Shell usage.

In the filtering section of the query in the next samplet, we're looking for CLOUD-SHELL in the ResourceGroup data column, but then digging even deeper to get more accurate results by ensuring the activity is related to a successful *Start* of storage creation. Anytime Cloud Shell is executed, storage is created in Azure.

```
| where ResourceGroup startswith "CLOUD-SHELL" //filtering for Cloud Shell  
| where ResourceProviderValue == "MICROSOFT.STORAGE" //To not mistake this for some other Cloud  
Shell operation, also filtering on MICROSOFT.STORAGE. Storage is created anytime Cloud Shell runs.  
| where ActivityStatusValue == "Start" //Making sure that the activity is the spawning of a new  
Cloud Shell instance
```

The next thing we need to do is determine how many times the individual that has been captured has run Cloud Shell. We do this with the [Summarize operator as covered in part/chapter 11](#).

```
| summarize count() by TimeGenerated , ResourceGroup , Caller , CallerIpAddress ,  
ActivityStatusValue //Getting a count of how many times each individual has run Cloud Shell
```

The last thing we need to do is take a couple pieces of important information and assign them as Entities. Entities are important for investigations. Without Entities, such as users, IP addresses, hostnames, file hashes, etc. we would have no evidence, or no clues with which to progress through an actual investigation. There are different ways to do this in the Analytics Rule wizard in Microsoft Sentinel, but you can also assign Entities in your KQL query by using the Extend operator to create custom data views – as covered in [part/chapter 13](#).

Microsoft Sentinel allows for four different custom entities in the queries. Those are:

- **AccountCustomEntity** – the user
- **IPCustomEntity** – the IP Address
- **HostCustomEntity** – the host (computer/device)
- **URLCustomEntity** – the capture URL the user accessed

Now, there's a much deeper discussion that can be had on Entities because much has changed (*and is constantly changing*) for this area in Microsoft Sentinel. See the following for more:

- [Microsoft Sentinel entity types reference](#)
- [Classify and analyze data using entities in Microsoft Sentinel](#)

But, for our purposes of learning KQL and applying the series' knowledge, let's stick with custom entities. Shown in the example below, we are assigning the known data columns of Caller (user name) and CallerIpAddress (user's IP) to the custom entities. This will capture the user and the user's IP address and place them in the Entities list associated with the Microsoft Sentinel Incident once an alert is generated based on our KQL logic.

```
| extend AccountCustomEntity = Caller //Assigning the Caller column - name of person - to AccountCustomEntity - this is what is used for the User Entity in Microsoft Sentinel Incidents  
| extend IPCustomEntity = CallerIpAddress //Assigning the CallerIpAddress column - IP Address of user's system - to IPCustomEntity - this is what is used for the IP Entity in Microsoft Sentinel Incidents
```

So, our full and complete KQL query to use when creating the Analytics Rule is:

```
AzureActivity //the table - this is where Cloud Shell activity is logged  
| where ResourceGroup startswith "CLOUD-SHELL" //filtering for Cloud Shell  
| where ResourceProviderValue == "MICROSOFT.STORAGE" //To not mistake this for some other Cloud Shell operation, also filtering on MICROSOFT.STORAGE. Storage is created anytime Cloud Shell runs.  
| where ActivityStatusValue == "Start" //Making sure that the activity is the spawning of a new Cloud Shell instance  
| summarize count() by TimeGenerated , ResourceGroup , Caller , CallerIpAddress , ActivityStatusValue //Getting a count of how many times each individual has run Cloud Shell  
| extend AccountCustomEntity = Caller //Assigning the Caller column - name of person - to AccountCustomEntity - this is what is used for the User Entity in Microsoft Sentinel Incidents  
| extend IPCustomEntity = CallerIpAddress //Assigning the CallerIpAddress column - IP Address of user's system - to IPCustomEntity - this is what is used for the IP Entity in Microsoft Sentinel Incidents
```

Use the following instructions to create an Analytics Rule with this query: [Create custom analytics rules to detect threats](#)

And if someone in your environment has run Cloud Shell recently, an alert and Incident will be generated that looks similar to the following:



The screenshot shows the Microsoft Sentinel interface for an incident titled 'Cloud Shell Execution' (Incident ID: 3942). The interface is divided into several sections:

- Header:** Shows the incident name, ID, and a 'Refresh' button.
- Left Sidebar:**
  - Description:** 'Keep track of when Cloud Shell is run and who did it.'
  - Alert product names:** 'Microsoft Sentinel'.
  - Evidence:** Shows 1 Event, 1 Alert, and 0 Bookmarks.
  - Last update time:** 02/17/22, 06:03 AM.
  - Creation time:** 02/17/22, 06:03 AM.
  - Entities (2):** 'rodrent@simillion...' and '23.96.34.207'.
  - Tactics and techniques:** 'Incident workbook' and 'Incident Overview'.
  - Analytics rule:** 'Cloud Shell Execution'.
  - Tags:** 'Cloud Shell'.
- Timeline:** A central pane showing a single event on Feb 16 at 3:33 PM: 'Cloud Shell Execution' (Low severity, Detected by Microsoft Sentinel, Tactics: PreAttack).
- Right Sidebar:**
  - Description:** 'Keep track of when Cloud Shell is run and who did it.'
  - Severity:** Low.
  - Status:** New.
  - Events:** 'Link to LA'.
  - Product name:** 'Microsoft Sentinel'.
  - Entities (2):** 'rodrent@simillion...' and '23.96.34.207'.
  - Tactics and techniques:** 'System alert ID: 293138fe-df48-687b-bc79-4...' and 'Rule name: Cloud Shell Execution'.
  - Last update time:** 02/17/22, 06:03 AM.
  - Updates:** 0.
  - Start time:** 02/16/22, 03:33 PM.
  - End time:** 02/16/22, 03:33 PM.
  - Alert link:** '--'.

Hey, look! Cloud Shell!



OK. Here's our very last extra credit together for this series and it's a reminder of some other things we've done together along the way. There's a bit more that we can do with this KQL query and Analytics Rule to make it a bit more intelligent. What if there are certain *"trusted"* people in our organization who should be able to run Cloud Shell without being captured as a potential suspect?

By creating a Watchlist (see: [Create watchlists in Microsoft Sentinel](#)) and modifying our KQL query slightly, we can ensure that only those individuals who *shouldn't* be able to run Cloud Shell are the only ones captured in our alerts.

How do we do that? Let's think back to [part/chapter 17 for the Let statement and the section on Creating Variables from Microsoft Sentinel Watchlists](#) and [part/chapter 8](#) when we discussed the allowable string and numeric predicates for the Where operator.

The following example shows those adjustments. I have a Watchlist in my environment called *TrustedUsers* that has a data column called *Username*. I maintain this Watchlist so that it contains the most current list of trusted users.

```
let watchlist = (_GetWatchlist('TrustedUsers') | project Username); //Putting the Usernames from our
Watchlist into memory to use later
AzureActivity //the table - this is where Cloud Shell activity is logged
| where Caller !in (watchlist) //filtering out our trusted users
| where ResourceGroup startswith "CLOUD-SHELL" //filtering for Cloud Shell
| where ResourceProviderValue == "MICROSOFT.STORAGE" //To not mistake this for some other Cloud
Shell operation, also filtering on MICROSOFT.STORAGE. Storage is created anytime Cloud Shell runs.
| where ActivityStatusValue == "Start" //Making sure that the activity is the spawning of a new
Cloud Shell instance
| summarize count() by TimeGenerated , ResourceGroup , Caller , CallerIpAddress ,
ActivityStatusValue //Getting a count of how many times each individual has run Cloud Shell
| extend AccountCustomEntity = Caller //Assigning the Caller column - name of person - to
AccountCustomEntity - this is what is used for the User Entity in Microsoft Sentinel Incidents
| extend IPCustomEntity = CallerIpAddress //Assigning the CallerIpAddress column - IP Address of
user's system - to IPCustomEntity - this is what is used for the IP Entity in Microsoft Sentinel
Incidents
```

=====