Microsoft

The Definitive Guide to KQL

Using Kusto Query Language for operations, defending, and threat hunting

Foreword by

Ann Johnson,
Corporate Vice President, Security, Microsoft

Mark Morowczynski • Rod Trent • Matthew Zorich

Sample files on the web

Microsoft

The Definitive Guide to KQL: Using Kusto Query Language for operations, defending, and threat hunting

Mark Morowczynski
Rod Trent
Matthew Zorich

The Definitive Guide to KQL: Using Kusto Query Language for operations, defending, and threat hunting

Published with the authorization of Microsoft Corporation by:

Pearson Education, Inc.

ISBN-13: 978-0-13-829338-3
ISBN-10: 0-13-829338-4

Library of Congress Control Number: 2024935858

$PrintCode

Trademarks

Warning and Disclaimer

Special Sales

For information about buying this title in bulk quantities, or for special sales opportunities (which may include electronic versions; custom cover designs; and content particular to your business, training goals, marketing focus, or branding interests), please contact our corporate sales department at corpsales@pearsoned.com or (800) 382-3419.

For government sales inquiries, please contact governmentsales@pearsoned.com.

For questions about sales outside the U.S., please contact intlcs@pearson.com.

EDITOR-IN-CHIEF
Brett Bartow

EXECUTIVE EDITOR
Loretta Yates

ASSOCIATE EDITOR
Shourav Bose

DEVELOPMENT EDITOR
Rick Kughen

MANAGING EDITOR
Sandra Schroeder

SENIOR PROJECT EDITOR
Tracey Croom

COPY EDITOR
Rick Kughen

INDEXER
Timothy Wright

PROOFREADER
Barbara Mack

TECHNICAL EDITOR
Corissa Koopmans

EDITORIAL ASSISTANT
Cindy Teeters

INTERIOR DESIGNER
codeMantra

COVER DESIGNER
Twist Creative, Seattle

COMPOSITOR
codeMantra

GRAPHICS
codeMantra

Dedication

For my friends and family, who I don't get to see nearly enough, particularly all my aunts, uncles, and cousins. And to all the defenders out there keeping the world safe. Thank you.

—Mark

For my beautiful wife, Megan, my wonderful kids, and my grandkids, Reid and Meredith, for all your patience, love, and support while allowing me time to focus on an important topic. And I would be remiss if I didn't also say thanks to all fans of KQL for your support in seeing KQL get its official Microsoft Press stamp.

—Rod

For my family, Megan, Lachlan, and Matilda, for all your patience, love, and support while I was writing this book.

—Matt

For my newborn son and my mother, who allowed me the time to work with these great authors and proof queries while my son slept.

—Corissa

Contents at a Glance

Contents

Acknowledgments

We would like to express our sincere gratitude to all the people who have supported us while writing this book. Without their help and encouragement, this book would not have been possible. This also includes the folks at Pearson/Microsoft Press: Loretta Yates and Shourav Bose for believing there was an audience for this and keeping us on track, and Rick Kughen, who turned our drafts into a book you are reading!

The reach of KQL in the Microsoft ecosystem is broader and more complex than any three people could possibly hope to cover. We would like to thank our colleagues, who have shared their expertise and insights on various operations and cybersecurity topics with KQL. There were so many great suggestions we couldn't even fit them all in the chapters, but they all made it to the GitHub repository. We'd like to graciously acknowledge the help and assistance from the people at Microsoft:

Estefani Arroyo, Michael Barbush, Kristopher Bash, Bailey Bercik, Keith Brewer, Chad Cox, Jack Davis, Varun Dhawan, Michael Epping, Marius Folling, Cosmin Guilman, Tim Haintz, Franck Heilmann, Mark Hopper, Laura Hutchcroft, Jef Kazimer, Corissa Koopmans, Gloria Lee, Michael Lindsey, Rudnei Oliveira, Razi Rais, Yong Rhee, Sravani Saluru, and Krishna Venkit.

We'd also like to give a special thanks to our Microsoft colleagues, Tarek Dawoud, who has provided us with overall valuable feedback, challenges, and suggestions on how to fully demonstrate KQL across Microsoft products; Mark Simos for his amazing graphics that simplify complex topics; and Aviv Yaniv for answering our numerous questions about several of the KQL language underpinnings.

We'd like to thank Ann Johnson for writing the foreword and her tireless leadership at Microsoft and in the information security industry. Security is truly a "team sport," and we are grateful to have you on our team.

Special thanks to Corissa Koopmans, our technical reviewer, who has been with us from the very start, going above and beyond multiple times throughout this book by challenging us, offering suggestions, and being willing to run through more queries than you can even imagine. We cannot thank you enough for your time, effort, and support throughout this entire process. Any mistakes in the book are solely because of the authors.

We want to thank you, the reader, for your interest and KQL curiosity. Our goal with this book was twofold. We want you to improve your environment's security posture and operations and add KQL to your professional skill set. When you finish this book,

you'll find that you are actually just beginning! We also hope it will inspire you to explore further, discovering new ways to continue improving the profession. We welcome your feedback and comments, and if you write a great query, tell us. We look forward to hearing from you!

Mark Morowczynski
Seattle, Washington

Rod Trent
Middletown, Ohio

Matthew Zorich
Perth, Western Australia

About the Authors

Mark Morowczynski is a principal product manager on the Security Customer Experience Engineering (CxE) team at Microsoft. He spends most of his time working with customers on their deployments in the Identity and Access Management (IAM) and information security space. He's spoken at various industry events, including Black Hat, Defcon Blue Team Village, Blue Team Con, Microsoft Ignite, and several BSides and SANS Security Summits. He has a BS in computer science, an MS in computer information and network security, and an MBA from DePaul University. He also has an MS in Information Security Engineering from the SANS Technology Institute. He can be found online on Mastodon at *@markmorow, @infosec.exchange* or his website at *markmorow.com*.

Rod Trent is a senior program manager at Microsoft, focused on cybersecurity and AI. He has spoken at many conferences over the past 30-some years and has written several books, including *Must Learn KQL: Essential Learning for the Cloud-focused Data Scientist*, and thousands of articles. He is a husband, dad, and first-time grandfather. In his spare time (if such a thing does truly exist), you can regularly find him simultaneously watching *Six Million Dollar Man* episodes and writing KQL queries. Rod can be found on LinkedIn and X (formerly Twitter) at *@rodtrent*.

Matthew Zorich was born and raised in Australia and works for the Microsoft GHOST team, which provides threat-hunting oversight to many areas of Microsoft. Before that, he worked for the Microsoft Detection and Response Team (DART) and dealt with some of the most complex and largest-scale cybersecurity compromises on the planet. Before joining Microsoft as a full-time employee, he was a Microsoft MVP, ran a blog focused on Microsoft Sentinel, and contributed hundreds of open-source KQL queries to the community. He is a die-hard sports fan, especially the NBA and cricket.

Foreword

Data is ubiquitous—generated by and flowing between applications, devices, users, and systems. It can provide valuable insights into the performance, behavior, and security of one's environment. However, accessing, analyzing, and acting on this data can be challenging. How can you turn it into actionable intelligence that can help optimize operations, enhance security, and solve problems?

One solution is KQL—Kusto Query Language—a powerful and expressive language that enables the querying and manipulation of large volumes of data in Azure Data Explorer, Azure Monitor, Azure Sentinel, and other Microsoft data platforms. KQL can help perform complex queries, apply advanced functions, and leverage operators to transform data into meaningful information. KQL can also help visualize data, create dashboards, and automate workflows.

KQL is critical for a modern cybersecurity team. It allows defenders to detect and respond to threats, anomalies, and incidents in near real-time. Whether a beginner or an expert, this book will teach everything readers need to know about KQL, including the fundamentals of the language, such as its syntax, functions, and operators. Readers will also learn how to write efficient and effective queries and manipulate and transform data.

In the later chapters, this book covers common security investigations using KQL and recommendations on leveraging KQL queries before these incidents occur. Readers will see these queries are just the beginning of what is possible with KQL. In the concluding chapter, the authors offer perspective on contributing their own KQL queries to the community, supporting the "team sport" of security.

This book is based on the experience and expertise of Mark, Matt, and Rod, Microsoft employees and KQL experts. They have authored this book to help individuals master KQL and to help organizations use the technology to improve their operational and security posture with data. Readers will also benefit from the additional queries and content contributed by different product managers, service engineers, and cloud solution architects who use KQL daily.

Readers will find this to be a practical guide—enabling readers to follow along, run included queries in their own environment, or use the sample datasets provided by the authors and help apply learnings.

Introduction

"Attacks always get better; they never get worse" (Schneier, 2011, para. 4).

Digital transformation has hit every large and small business in the world. If you were born before the year 2000 and look at how you book travel, order food, and find tickets for an event today, you will realize the methods and technologies you use are much better than they once were. They are much more digitized and often provided by very different vendors. The cloud has brought this disruption to the market of ideas and innovation at a global scale. This digital transformation of our world has been very disruptive to all industries and organizations, causing cloud adoption at an unprecedented scale. Adopting the cloud is no longer seen as a luxury or a thought experiment. It is imperative to remain competitive and relevant as a business. It has fundamentally shifted the way a business operates.

This business shift has impacted how IT professionals, information security professionals, and even developers work day to day. Operational IT staff no longer just have on-premises servers to manage. Their responsibilities have increased and changed dramatically with the shift to the cloud. Servers can now operate entirely in the cloud, and cloud-native platform as a service (PaaS) or software as a service (SaaS) solutions form a significant part of many companies' system portfolios. These systems' performance, availability, and resilience are more crucial than ever.

Understanding big data analytics concepts now impacts IT operations staff in many facets of their day-to-day work. IT professionals and developers can now scale up or scale out resources and deploy code changes multiple times a day to meet the needs of their business. With this comes the need for telemetry to make those operational decisions.

For information security professionals, the change is even more drastic. There is now more of everything. There are more organizational resources than ever before. More users are accessing these resources from more devices and more locations. There are just more things to monitor malicious activity for. It used to be the goal to have a Security Incident & Event Management (SIEM) system that integrates and pulls data from all sources. However, your security team is now swimming in data. Being able to sift through data masterfully and quickly is now your primary challenge. Adversaries are aided with automatic tools to perform more attacks, leading many companies to adopt a Zero Trust framework.

Assume breach is a core tenant of Zero Trust, creating a shift in the modernization of organizational security operations. We are drowning in raw data. Organizations need to focus on managing realized risk—risk that has actually happened—and need to take action on this risk quickly. Serious cyberattacks are often driven in near-real-time by human attack operators.

This is why a core metric of a modern security operations team should be 'mean time to remediate' (MTTR). How quickly did we detect the attacker and stop them from meeting their goals? In other words, how did we reduce attacker dwell time? The less time the attacker has to conduct their operation results in less time the attacker can cause damage, reducing organization risk.

But how do organizations speed up this detection process with all this data? The answer is moving from raw data ingestion as a traditional Security Incident and Event Management (SIEM) to a more automated approach on actionable insights using Security Orchestration, Automation, and Remediation (SOAR) technologies and integrating toolsets. Figure 1 depicts modern security operations capabilities.

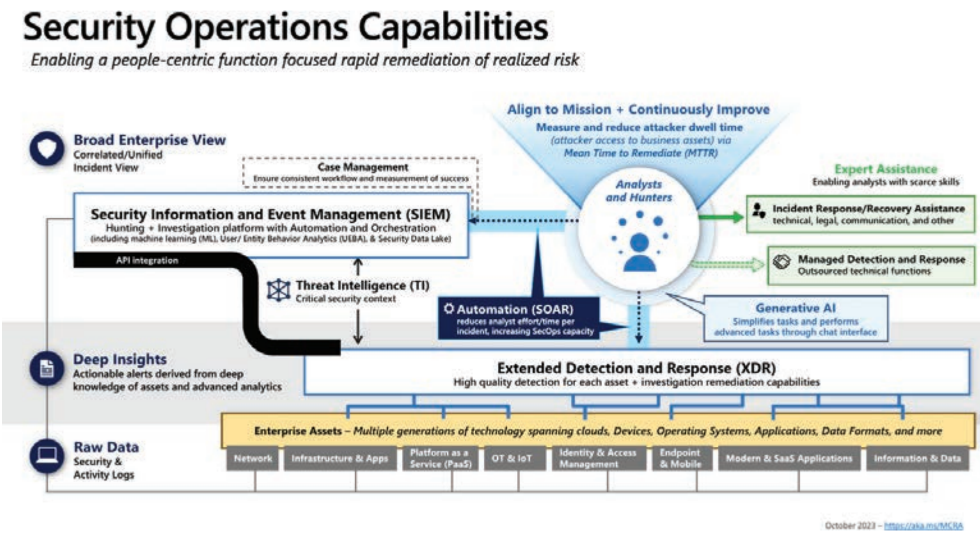


FIGURE 1 Turning raw data into insights and action of a modern SOC

SOAR has a few benefits for analysts and threat hunters. First, manual work should be reduced. Instead of spending time moving between different tools and consoles, connecting data points together in different languages, more meaningful work is being done, fighting the adversary. Second, because automation is happening at machine speed rather than human speed, our response times are greatly speeding up. Finally, our analysts and hunters can handle this increase in the scale of the environment, including the growing number of attacks taking place both in scope and complexity.

This leads us to why you've picked up this book. The language you will use to unlock these actionable insights and detect the most advanced attacks as part of SOAR is the Kusto Query Language, better known as KQL, which is at the heart of the Microsoft cloud for parsing data from various datasets. You will be able to quickly search through millions of records across multiple products to determine the scope and detect some of the most advanced attacks. More importantly, you will take action to remediate it natively in tools like Microsoft Sentinel and Microsoft Defender.

The KQL language must become second nature for information security professionals, just as PowerShell or Python is today. Microsoft's latest threat actor detections found in blog posts and playbooks and community-shared detections include KQL queries. These need to be run, modified, and adapted for your environment to continue driving down that MTTR (mean time to repair) in an ever-growing environment. Every second counts.

> **Note** The full Microsoft Cybersecurity Reference Architecture and more can be found at *aka.ms/mcra*.

Organization of This Book

This book is divided into six chapters, moving from the basics and most common KQL tasks you will perform. Chapter 1, "Introduction and Fundamentals," and Chapter 2, "Data Aggregation," introduce the basics. Chapter 3, "Unlocking Insights with Advanced KQL Operators," and Chapter 4, "Operational Excellence with KQL," introduce more advanced functionality and begin putting the power of KQL into practice. The final chapters, Chapter 5, "KQL for Cybersecurity," and Chapter 6, "Advanced KQL Cybersecurity Use Cases and Operators," delve into defending and threat hunting and how the skills learned throughout this book can be used from a security perspective.

Each chapter is self-contained and tries to be as independent as possible so they can be read individually. However, there are cross-references between chapters, so you might sometimes need to read a section in a different chapter to get the big picture.

We tried to make this book accessible for a broad range of people with varying KQL expertise, including those who are leveraging the skills taught here for the first time, as well as those who have been using KQL for many years. If you are new to KQL, start with Chapter 1 and work your way forward. If you are a seasoned KQL expert, quickly skim the first two chapters before diving into the more advanced topics.

Who Should Read This Book?

This book is for anyone leveraging Microsoft cloud resources such as the Azure or Microsoft 365 suite of products, including administrators, engineers, architects, and even developers who want to be able to monitor and understand what is happening in their environment and then use those insights to take action to improve the environment. It's also for information security professionals who can monitor and take action on malicious activity as quickly and efficiently as possible.

Conventions and Features in This Book

This book presents information using conventions designed to make the information readable and easy to follow.

- Sidebar elements with labels such as "Note," "Tip," or "Caution" provide additional information. Many Tips provide queries from Microsoft professionals, which you can use in your environment.
- Text that you type (apart from code blocks) appears in bold.
- A plus sign (+) between two key names means that you must press those keys at the same time. For example, "Press Alt+Tab" means that you hold down the Alt key while you press the Tab key.
- A chevron—>—between two commands (e.g., File > Close) means that you should select the first menu or menu item, then the next, and so on.

System Requirements

Examples and scenarios in this book require access to an Azure Log Analytics environment and a computer that can connect to Azure using an up-to-date browser such as Microsoft Edge, Google Chrome, or Apple Safari. A demo Log Analytics environment is available at *aka.ms/LADemo*. For some advanced scenarios, we use Azure Data Explorer. See *dataexplorer.azure.com/clusters/help/databases/Samples*.

GitHub Repo

The book's GitHub repository includes all the KQL queries used throughout this book for easy copying and pasting as well as any of the sample datasets used in the chapters: *https://github.com/KQLMSPress/definitive-guide-kql*.

The download content will also be available on the book's product page at *MicrosoftPressStore.com/DefKQL/downloads*.

Errata, Updates, and Book Support

We've made every effort to ensure the accuracy of this book and its companion content. You can access updates to this book—in the form of a list of submitted errata and their related corrections—at:

MicrosoftPressStore.com/DefKQL/errata

If you discover an error that is not already listed, please submit it to us at the same page.

For additional book support and information, please visit *MicrosoftPressStore.com/Support*.

Please note that product support for Microsoft software and hardware is not offered through the previous addresses. For help with Microsoft software or hardware, go to *support.microsoft.com*.

Stay in Touch

Let's keep the conversation going! We're on X / Twitter: *twitter.com/MicrosoftPress*.

CHAPTER 2

Data Aggregation

After completing this chapter, you will be able to:

- Perform common statistical analysis on data such as counting totals, distinct counts, and the first and last time an event takes place
- Group your data by common time delimitations such as week, day, or hour
- Visualize your dataset in various graph types

We Are Dealing with a Lot of Data Here

In the previous chapter, we stressed how critical it is to filter down the initial starting data to your desired dataset. There were many ways to do this: by time, by specific values in a column, and by when a specific value was not present. Despite being able to filter down millions of records to a subset you want to look at, you're often left with, well, a lot of data—too much to deal with manually.

For example, let's say you work at a 45,000-user company based in Chicago. You have large offices in New York, Atlanta, and Seattle. You also have smaller offices in New Orleans and Denver and a few international offices in London, Paris, and Tokyo. A phishing message is sent to all your users. It's a very good message, and many of them fall for it. Your leadership team wants to know how many fell for it and which offices are impacted the most. You filter based on that specific message in the last 14 days and your heart drops; it's 12,139.

Reporting on that number to your leadership team isn't good enough. They need to know which office was most affected because the New York office has much of the finance team, and quarterly earnings will be posted in 10 days. The Chicago office is the home to the main research and development team. The Paris office is closing a strategic deal with a partner. Knowing which users at these locations are possibly compromised is critical because some parts of the business could suffer more impact if those compromises are not remediated quickly. With 12,139 users affected, that's far too many to sort into regions manually.

In an attempt to reduce the dataset, you apply another filter to those locations, and the number drops to 7,013. However, in the sign-in logs, you notice that the same user is shown three times because multiple sign-ins have occurred. How do you determine if the user or the threat actor did those sign-ins? You also still have too many users to determine which region was hit the hardest.

Your leadership team needs to give a status update to the company's senior leadership team. You have a few choices. First, you can just scroll down the list, trying to get a rough estimate based on what users you recognize. That is no way to make a critical and strategic decision. You can try exporting this data to another tool like Excel, where you can do additional deduplication filtering, but some data types don't export cleanly, so many of your tools won't work. So, to fully use the data export, more work must be done on those 7,013 records.

Or you can use another strength of KQL, data aggregation. In this chapter, we will show you how to answer these questions quickly and include much more information, such as the first and last time this was witnessed. You will turn your dataset into insights and actions. You can also convert them into one of the things managers love most: pretty charts. Many of the functions discussed in this chapter will be used as building blocks to answer questions like those in our scenario and many more!

Obfuscating Results

Before we jump into a whole chapter full of queries, you should know there are ways to enable auditing of your queries. We can skip the whole "with great power comes great responsibility" admin talk here. The important thing is knowing your query might show up in the audit logs.

Those queries might contain sensitive information, such as an API key/secret or possible personally identifiable information (PII) about a user. The good news is there is a very simple way to tell KQL to obfuscate the string. Simply add `h` or `H` before the string you are trying to match. Obfuscation will not work in our Log Analytics Demo environment, but this is a good habit to get into. The audit results are displayed in Figure 2-1.

```
"QueryTimeRangeStart": ,
"QueryTimeRangeEnd": ,
"QueryText": SigninLogs
| where TimeGenerated > ago (30d)
| where ResultType == 0
| where UserDisplayName has '***'
```

FIGURE 2-1 Query text that has been obfuscated in the audit logs

The query to obfuscate those strings is very simple:

```
SigninLogs
| where TimeGenerated > ago (30d)
| where ResultType == 0
| where UserDisplayName has h'mark.morowczynski'
```

Again, this will not work in our Log Analytics Demo environment, and none of the queries that we'll cover in this chapter have secret info or PII, but if you are slightly modifying these and running them in your production environment, add that `h` or `H` beforehand, so the strings would be obfuscated in the audit logs.

Distinct and Count

Some common scenarios you will need to repeat repeatedly are narrowing down to the distinct number of elements returned and counting the elements. Often, you'll want to combine those two things! We can do all that and much more.

Distinct

We'll start with the `distinct` operator, which will return the results based on the distinct combination of columns you provide. We'll start by trying to answer a simple question: How many different user agents are being used in the environment? If we run our query as we did in Figure 2-1, we'll see we have many different records; see Figure 2-2.

```
SigninLogs
| where TimeGenerated > ago (14d)
| project UserAgent
```

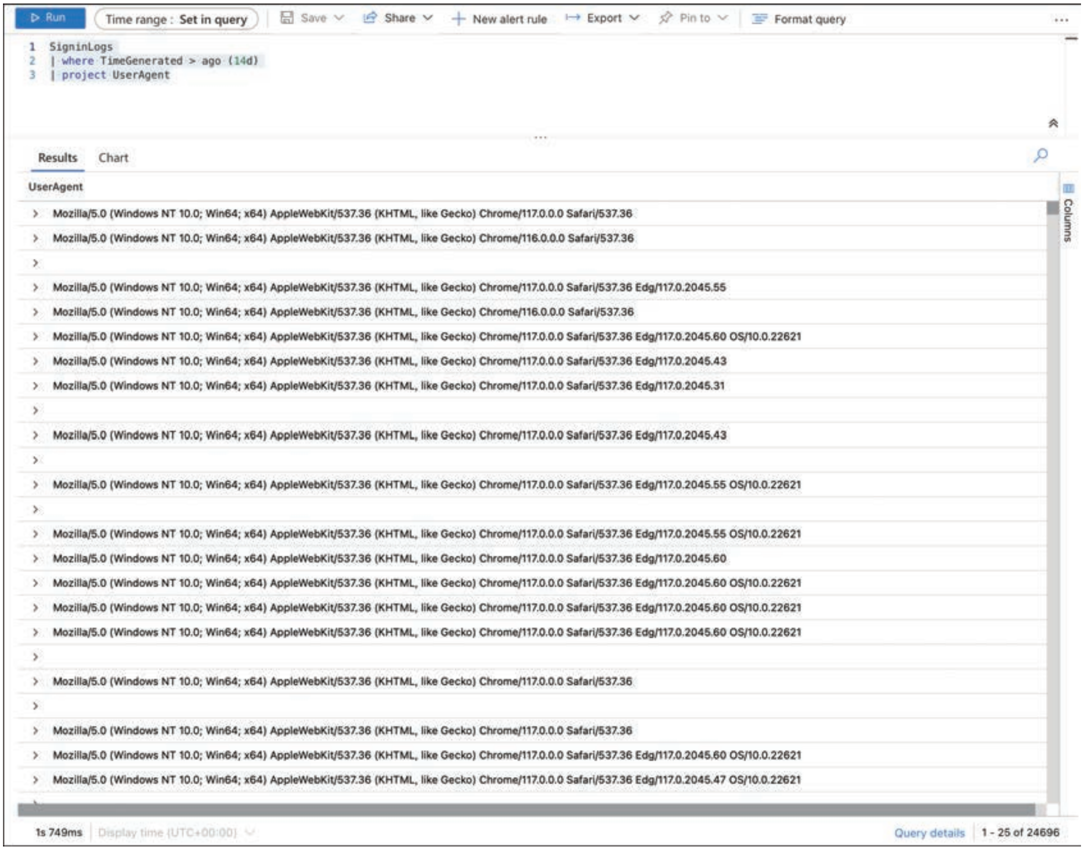


FIGURE 2-2 User agents that have been used in the last 14 days

As you can see, in the last 14 days, we had 24,696 sign-ins, and the list of the different user agents available seems pretty varied. The first two results are the same; if we look near the bottom, the third and fifth results are the same. But to answer our question, we need to remove the duplicates and only return unique values. Let's try our query again, but instead of using `project`, let's use the `distinct` operator in its place. The results should look similar to Figure 2-3.

```
SigninLogs
| where TimeGenerated > ago (14d)
| distinct UserAgent
```

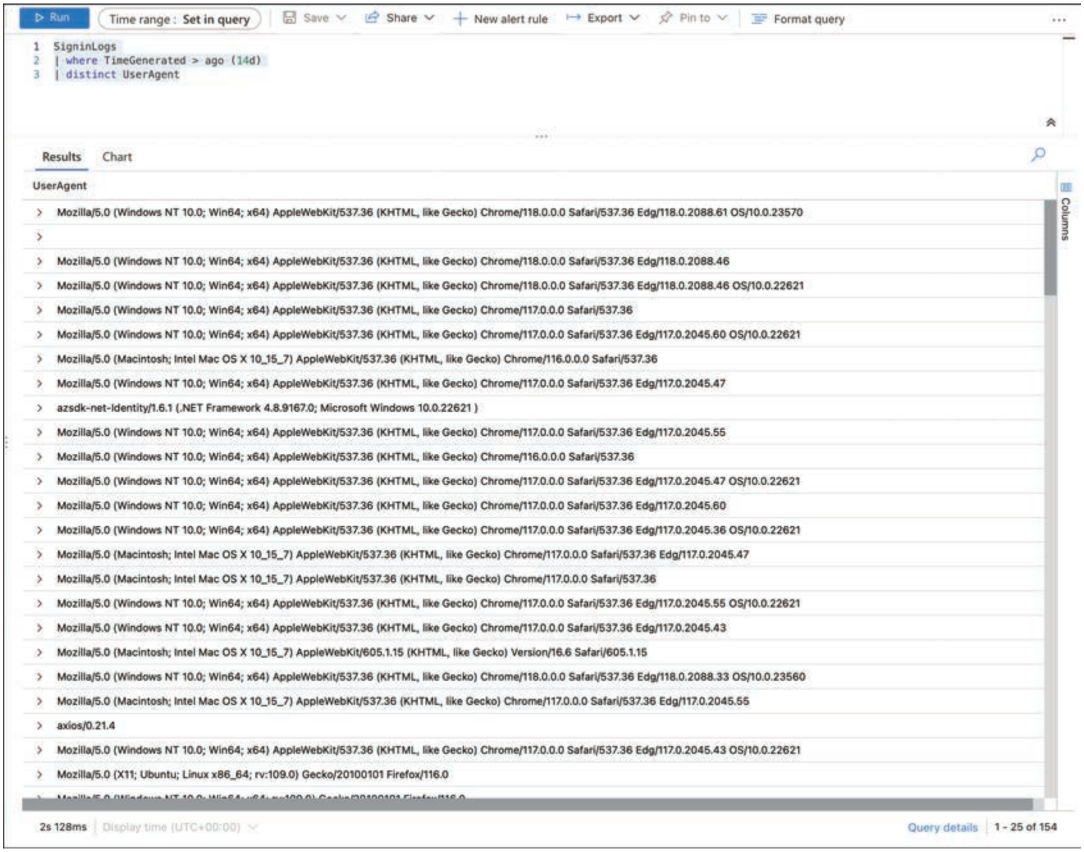


FIGURE 2-3 Distinct user agents that have been used in the last 14 days

Our dataset was further reduced to 154 unique `UserAgent` strings in this environment. We need to work on some of our device management and patching to reduce this number further and ensure that our environment is uniform. A few other things now easily stick out. First, the last row shows a user using Firefox on Ubuntu. Do our security policies and Microsoft Entra ID conditional access policies apply to the Linux platform? If not, we probably need to turn this insight into action and update our policies. Also, third from the bottom is the `axios/0.21.4` user agent. This looks very different from our other user agents. Is this expected in this environment? It's hard to say; this is a demo environment, so probably.

Looking through these types of results in your own data can lead to many interesting discoveries. Besides finding gaps in their Microsoft Entra ID conditional access policies, we've had customers find pockets of computers that were never upgraded to the latest operating system, running unpatched and unsupported in production. We can do a few other things to make important findings stand out a bit more, which we'll get to shortly.

The `distinct` operator isn't limited to one column. You can add multiple columns in your query and get the distinct values of that combination. Let's expand on the previous scenario, where we looked for the unique number of user agents being used and now extend it to which user agents are accessing which applications. We can easily update our query to include applications. Run the following query and add the sorting direction for clarity. Your query should look similar to Figure 2-4:

```
SigninLogs
| where TimeGenerated > ago (14d)
| distinct AppDisplayName, UserAgent
| sort by AppDisplayName asc
```

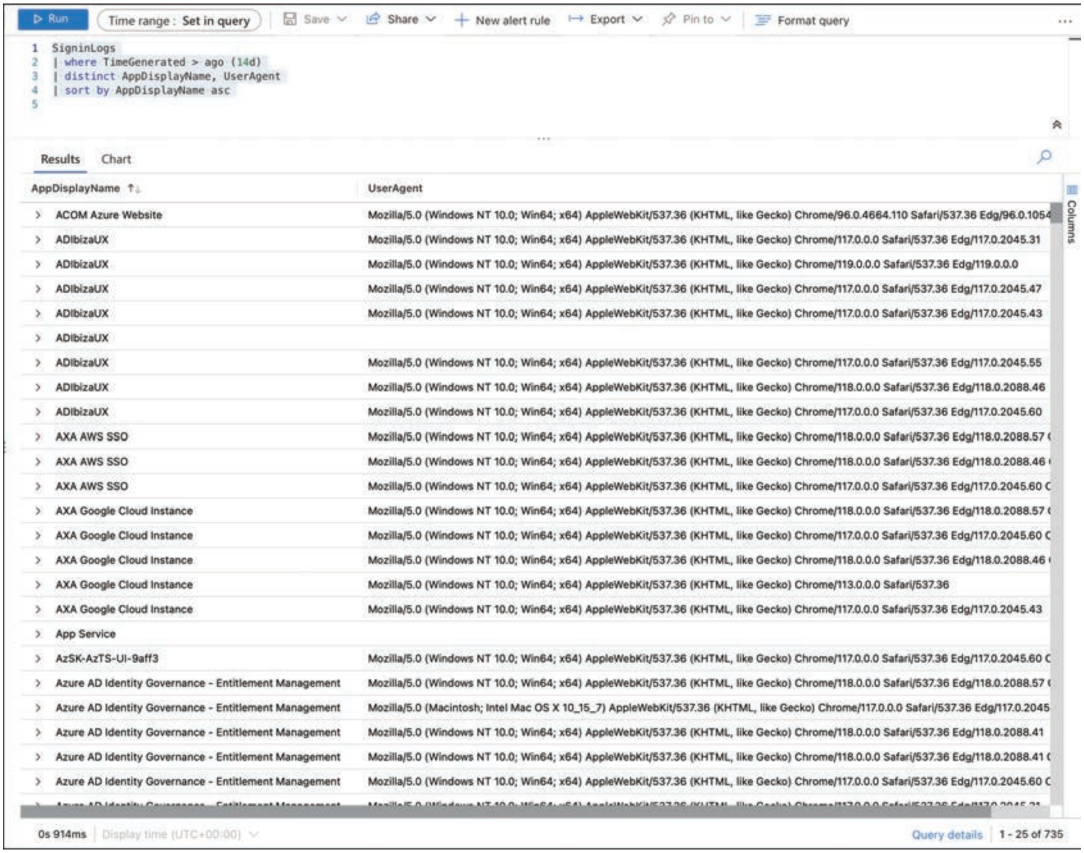


FIGURE 2-4 Distinct applications and the user agents accessing them

We can now tell the unique instance of each user agent mapped to which application they were accessing. About halfway down the screen, we see five different `UserAgent` strings used against the AXA Google Cloud Instance application. This is easy enough for us to see, and we can actually see one of those browsers is much older than the others: Chrome 113. But what if we also need to determine the count across all the applications and user agents/browsers?

Summarize By Count

Before we can answer that question directly, we need to introduce a new operator: `summarize`. We'll use this frequently in this chapter and the rest of the book. The `summarize` operator will summarize data and produce a table of the aggregated results. There are several `aggregate` values, such as `count()`, `dcount()`, `countif()`, and `dcountif()`, which we'll discuss in this section. We'll cover additional `aggregate` values later in this chapter, such as finding the minimum and maximum values.

The `summarize` operator follows an input pattern of first specifying a column name for the outputted results of the query you are about to run. This is optional; if nothing is chosen, the default name will be used. The second input is the name of the `aggregate` function you are using, such as `count` or `dcount`. The next output determines which column(s) you want passed through the `aggregate` function. That seems complicated, but you'll see shortly that this can be extremely powerful.

We'll start the first query with `summarize`, similar to what we did in the previous chapter, by selecting a random sample value—in this case, a table column—and pass it into the `aggregate` function. To do this, we will use the `take_any() aggregate` function. Note that `any()` has been deprecated. Run the following query; your output should be similar to Figure 2-5:

```
SigninLogs
| where TimeGenerated > ago (14d)
| project TimeGenerated, UserAgent, AppDisplayName
| summarize take_any(*)
```

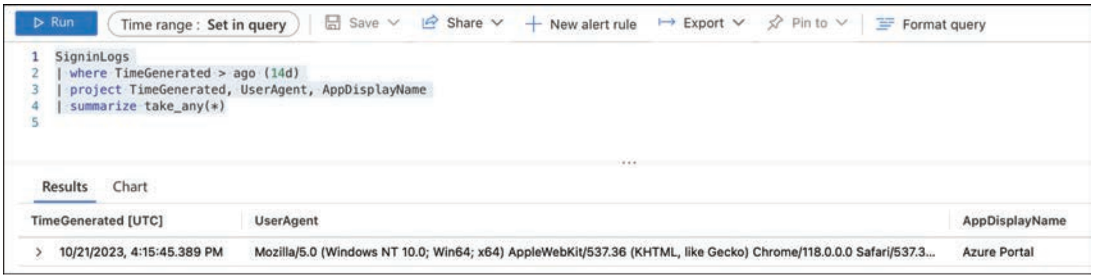


FIGURE 2-5 A random sample row has been returned

This query returned a random row, and we altered our output to show the `TimeGenerated`, the `UserAgent`, and `AppDisplayName` columns. If we wanted to see just the value for `UserAgent` with `summarize`, we could also do that by specifying that column in the `take_any()` function.

Tip This query is useful for operational and security teams alike. If you don't know which applications are currently leveraging WPAD, this will help you start to build that list. If you do see suspicious names or unexpected applications, these should be further investigated. –Michael Barbush, Senior Cloud Solution Architect

```
//Change timeframe to fit needs
DeviceNetworkEvents
| where RemoteUrl has 'wpad' and Timestamp > ago(1h)
| summarize by InitiatingProcessFileName, InitiatingProcessVersionInfoProductName,
  RemoteUrl, ActionType
| sort by InitiatingProcessFileName asc
```

Because we have a good handle on the `UserAgent` value, let's try and answer a question: Which `UserAgent` string values do we have in this environment, and how often do they show up? To do that, run the following query; your output should look similar to Figure 2-6.

```
SigninLogs
| where TimeGenerated > ago (14d)
| summarize count() by UserAgent
```

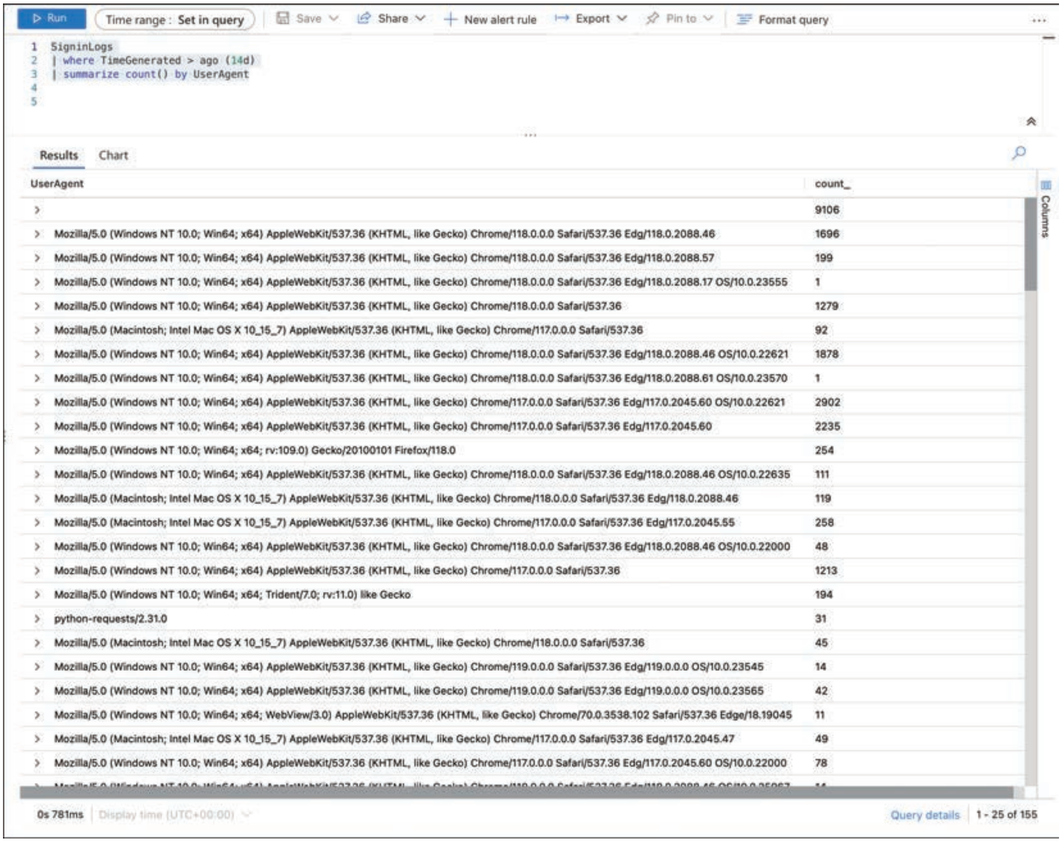


FIGURE 2-6 UserAgents by how many times they were found

Again, a few things should stick out. First, we didn't provide a column name for the `count()` aggregation, so it's just named `count_`. We can set that display value, which we will do in the next query. Second, we have a wide range of values for `count`. A good operational practice is to look at the longer tail of these results by looking at user agents that have only a handful of results, which might identify clients that need to be updated or an attacker that has misspelled a user agent name when trying to blend in with the normal traffic. Run the following query; the output will be similar to Figure 2-7.

```
SigninLogs
| where TimeGenerated > ago (14d)
| summarize UserAgentCount = count() by UserAgent
| sort by UserAgentCount asc
```

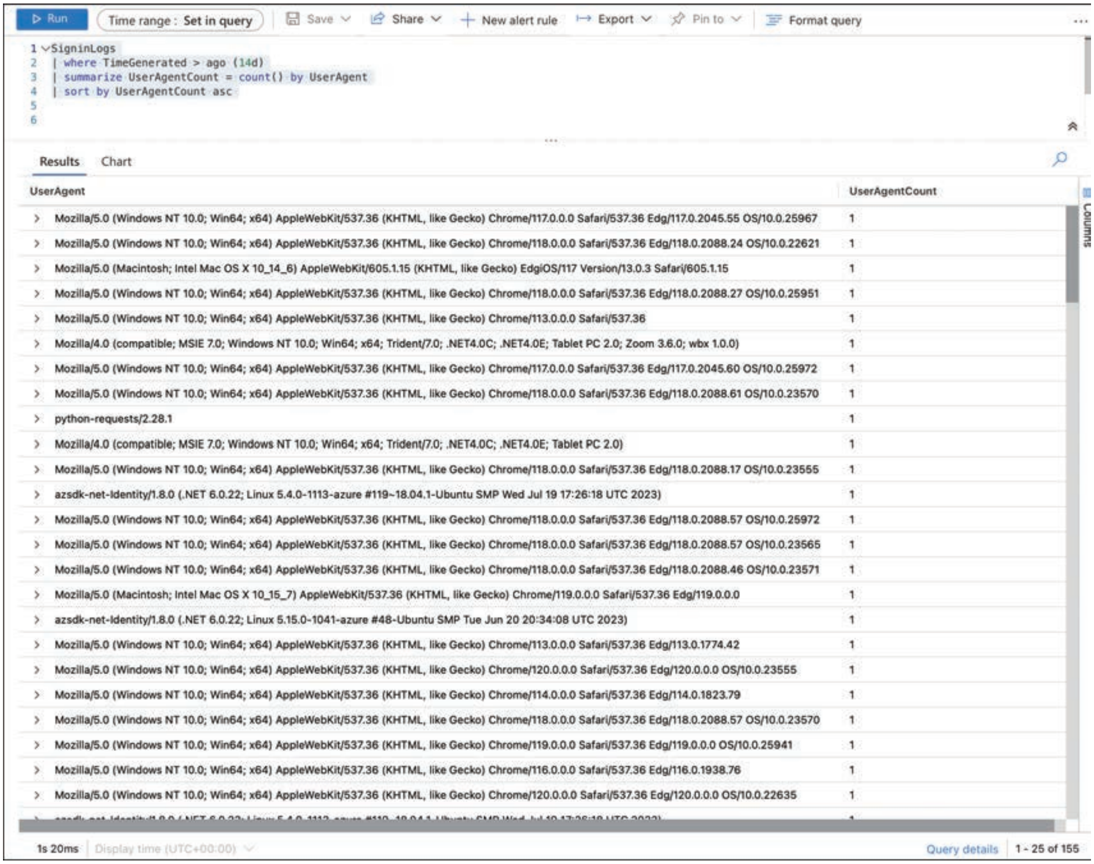


FIGURE 2-7 UserAgents by how many times they were found, sorted from least to most

Many user agents have only been seen once in the last 14 days. But `python-requests/2.28.1` sticks out; we should investigate it. We can add additional columns to the `count() by`. This will allow us to determine which user agent accessed each application. Run the following query; your output will be similar to Figure 2-8.

```
SigninLogs
| where TimeGenerated > ago (14d)
| summarize UserAgentCount = count() by UserAgent, AppDisplayName
| sort by UserAgent desc
```

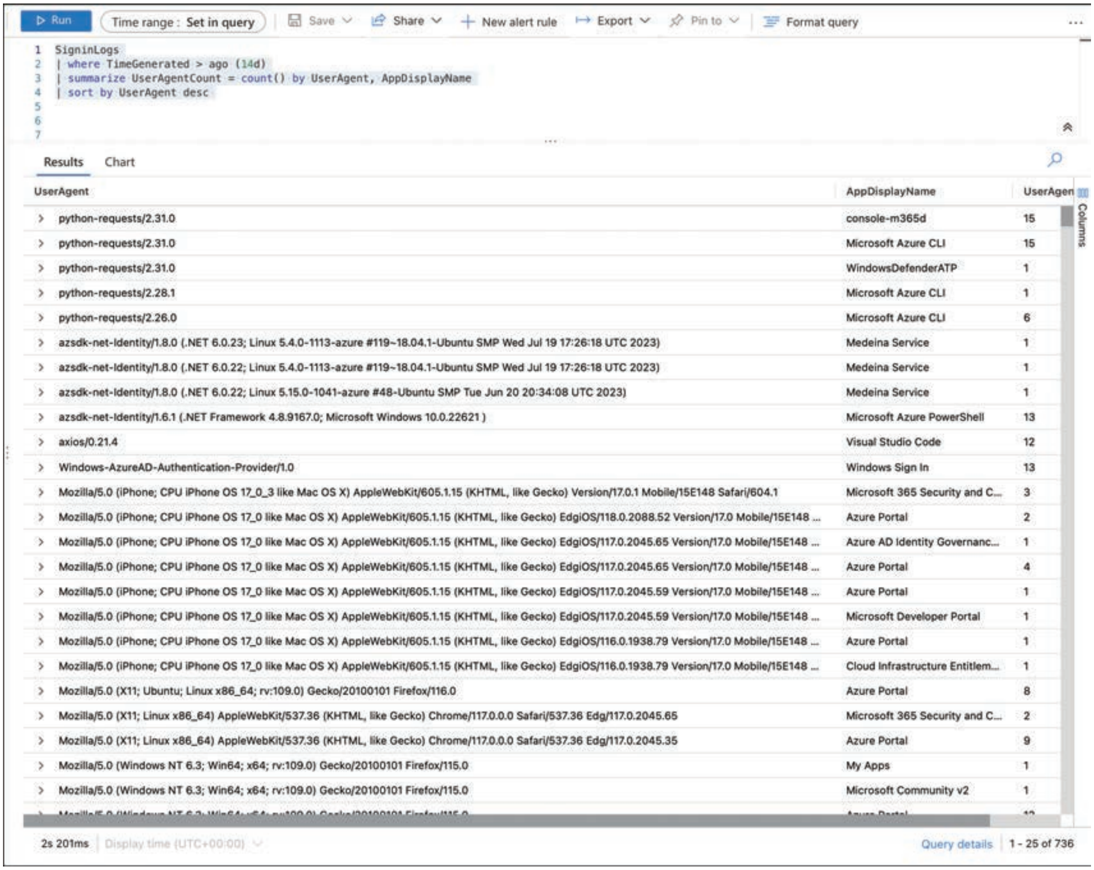


FIGURE 2-8 UserAgents Sorted Z to A with what apps they accessed

The `python-requests/2.28.1` request accessed the Microsoft Azure CLI application once. But even more interesting, we see other user agents named `python-requests` in this environment. Look to see what information you uncover in your environment.

Tip This query summarizes the count of API requests to Microsoft Graph APIs for a specific application, with metadata about the clients, such as IP Address and UserAgent strings. This can be useful to understand more about the deployment and use of a specific application in your tenant. The Location field reflects the region of the Microsoft Graph service that serves the request. This is typically the closest region to the client. –Kristopher Bash, Principal Product Manager

```
MicrosoftGraphActivityLogs
| where TimeGenerated > ago(3d)
| where AppId =='e9134e10-fea8-4167-a8d0-94c0e715bcea'
| summarize RequestCount=count() by  Location, IPAddress, UserAgent
```

We can also look at this query from the application perspective if we want to know which application has been accessed the most by which user agent. To determine this, we'll simply flip our `count() by`. Instead of counting by user agent, we'll count by application and show which user agent is accessing that application the most. Run the following query; your output should be similar to Figure 2-9.

```
SigninLogs
| where TimeGenerated > ago (14d)
| summarize AppDisplayNameCount = count() by AppDisplayName, UserAgent
| sort by AppDisplayNameCount desc
```

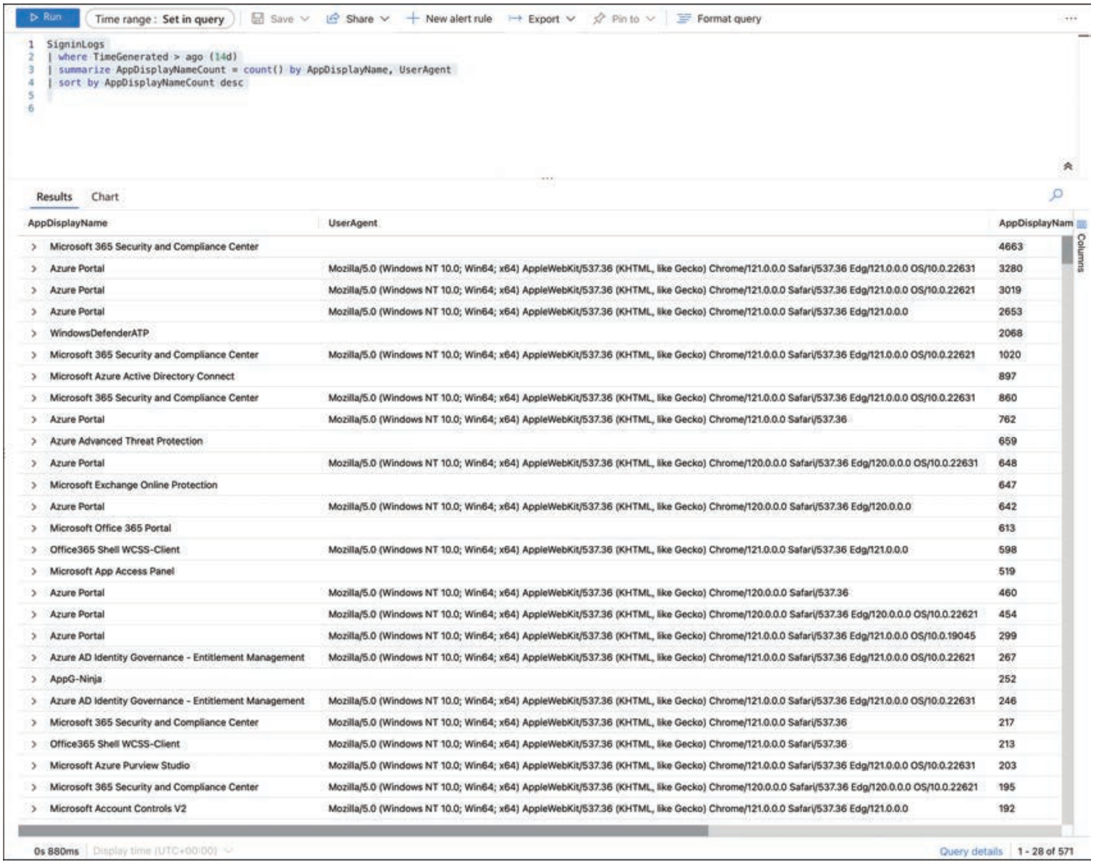


FIGURE 2-9 Most-accessed application by user agent

In this demo environment, the Azure Portal application with an Edge browser version 121.0.0.0 was used 2,653 times. At the start of this section, we focused on getting the distinct set of results returned, but we had to count manually. Then, we used a `count()` of the results returned, but these are not distinct. Let's combine both of these with the `aggregate` function `dcount()`, which allows us to get the estimated distinct count by passing the column for which we want to get a distinct count and which additional columns we want to aggregate/group the data by. Let's take our current example. What user agent is accessing the most unique applications? Run the following query; your output should be similar to Figure 2-10.

```
SigninLogs
| where TimeGenerated > ago (14d)
| summarize AppDisplayNameCount = dcount(AppDisplayName) by UserAgent
| sort by AppDisplayNameCount desc
```

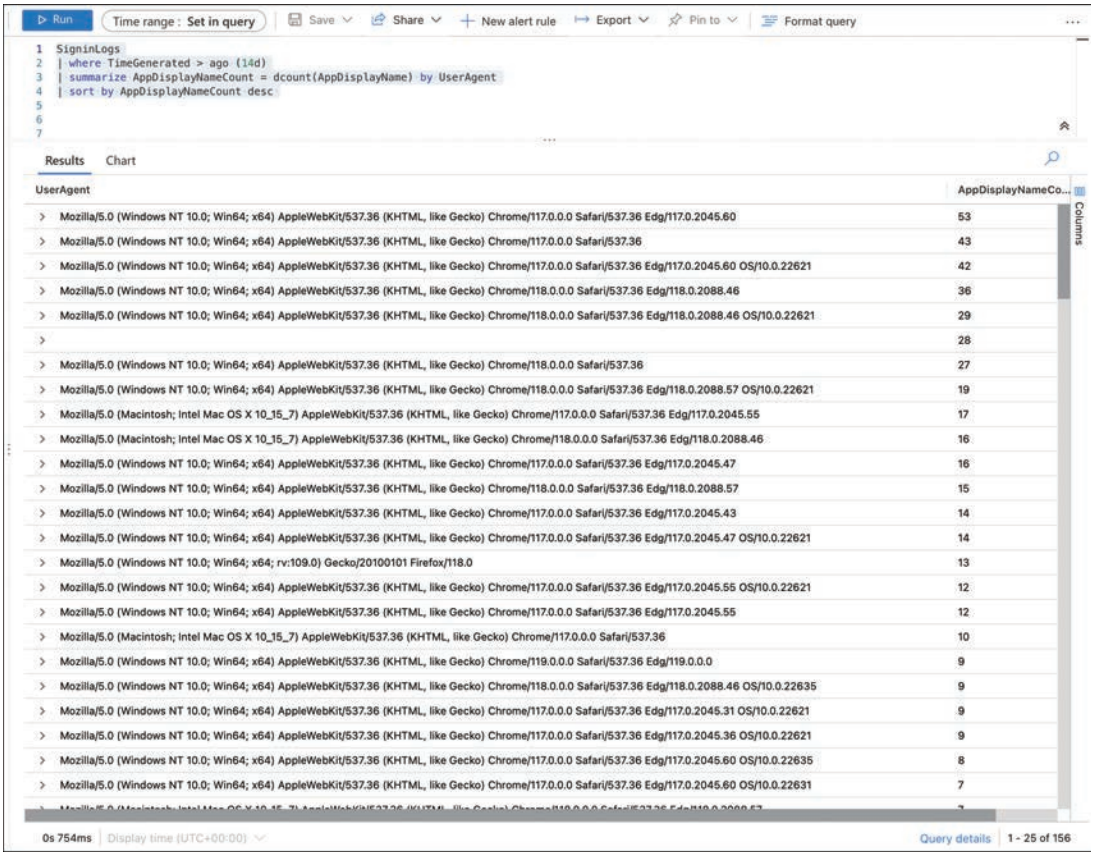


FIGURE 2-10 Distinct applications and how many times a user agent has accessed them

This is extremely useful information as we can see our most used user agent in the environment regarding the total number of applications it is accessing. Sorting the opposite way is also interesting to see what user agent is accessing only a small number of apps. These might be good candidates to be updated and brought into the standard browser versions for the environment.

Tip These queries offer critical insights into activities necessitating further scrutiny. This suite of queries is designed to enumerate operations linked to pivotal identity governance features, thereby illuminating the extent of Identity Governance and Administration (IGA) activities. It aims to enhance administrator awareness regarding configuration modifications and end-user actions, including access requests, approvals, and subsequent assignments. Further exploration of specific operations provides a deeper understanding of the access governance state, showcasing the efficiency of implemented access control measures. Ensure your query time range includes as much history as you have enabled for retention in your log analytics workspace. –Jef Kazimer, Principal Product Manager

```
AuditLogs
| where LoggedByService == "Entitlement Management"
| summarize OperationCount = count() by OperationName, AADOperationType
| order by OperationCount desc

AuditLogs
| where LoggedByService == "Access Reviews"
| summarize OperationCount = count() by OperationName, AADOperationType
| order by OperationCount desc

AuditLogs
| where LoggedByService == "Lifecycle Workflows"
| summarize OperationCount = count() by OperationName, AADOperationType
| order by OperationCount desc

AuditLogs
| where LoggedByService == "PIM"
| summarize OperationCount = count() by OperationName, AADOperationType
| order by OperationCount desc
```

We can also flip this. What if we want to see how many unique user agents access each application? We can see this number pretty quickly by getting the `dcount()` for the `UserAgent` column and grouping by application. Run the following query; your results should be similar to Figure 2-11:

```
SigninLogs
| where TimeGenerated > ago (14d)
| summarize UserAgentCount = dcount(UserAgent) by AppDisplayName
| sort by UserAgentCount desc
```

This is even more interesting; 100 different user agents access the Azure Portal! Thankfully, this is a test environment, but this tells a compelling story. Many customers will have their own line-of-business (LOB) applications in Microsoft Entra ID. Running a similar query and seeing many user agents will show the possible browsers that would need to be tested to ensure compatibility. That's great data for the leadership team to show why standardization on specific versions should be warranted.

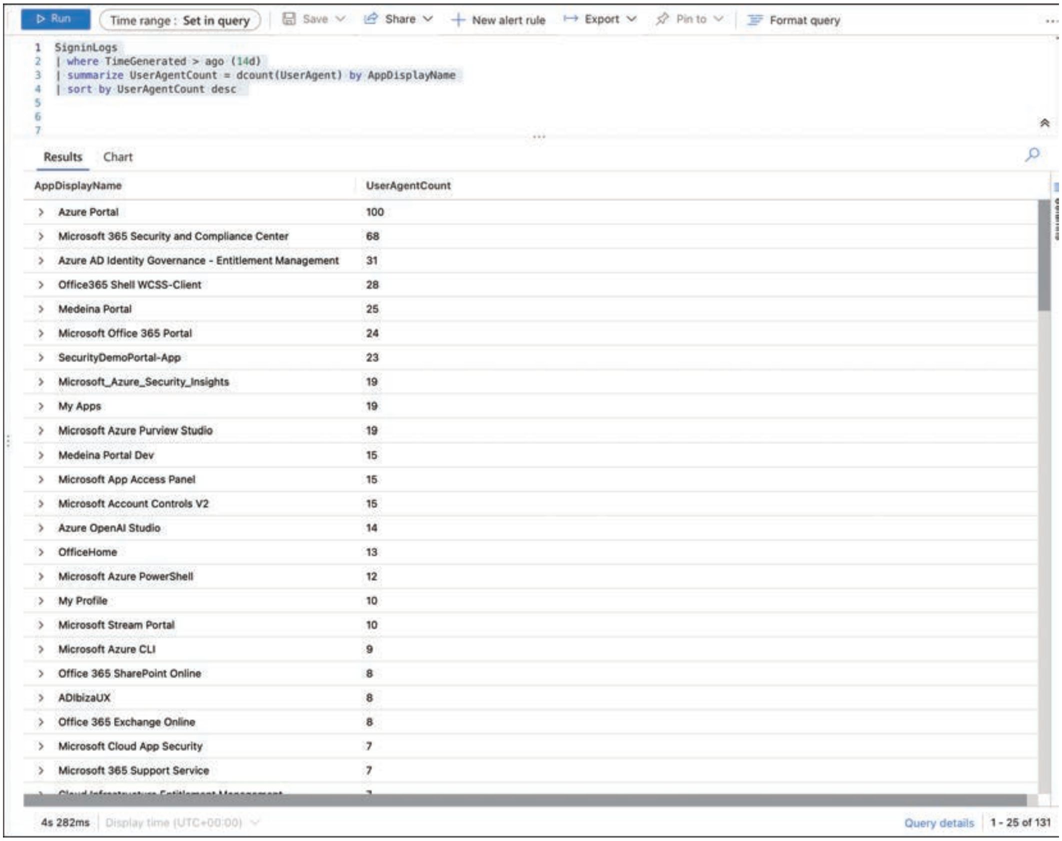


FIGURE 2-11 Counting the distinct user agents and which applications they accessed

> **Note** In the Log Analytics demo environment, `UserPrincipalName`, `UserID`, and `UserDisplayName` are blank. However, these are excellent columns for your queries when looking for unique things in your environment.

There are two other similar aggregation functions to `count` and `dcount`: `countif` and `dcountif`. These functions allow you to count the rows if the expression passed to it evaluates true. For example, we have many applications in our Microsoft Entra ID tenant. We want to be able to determine the number of access attempts per application, and we want to see how many occurred in the US region. You could accomplish this by running two separate queries, one for the total count and then another where you filter based on location. But with `countif`, you can accomplish this in one query and see the results side by side. Run the following query; your results should be similar to the output in Figure 2-12:

```
SigninLogs
| where TimeGenerated > ago(14d)
| summarize TotalCount = count(), USLogins=countif(Location == "US") by AppDisplayName
| sort by USLogins desc
```

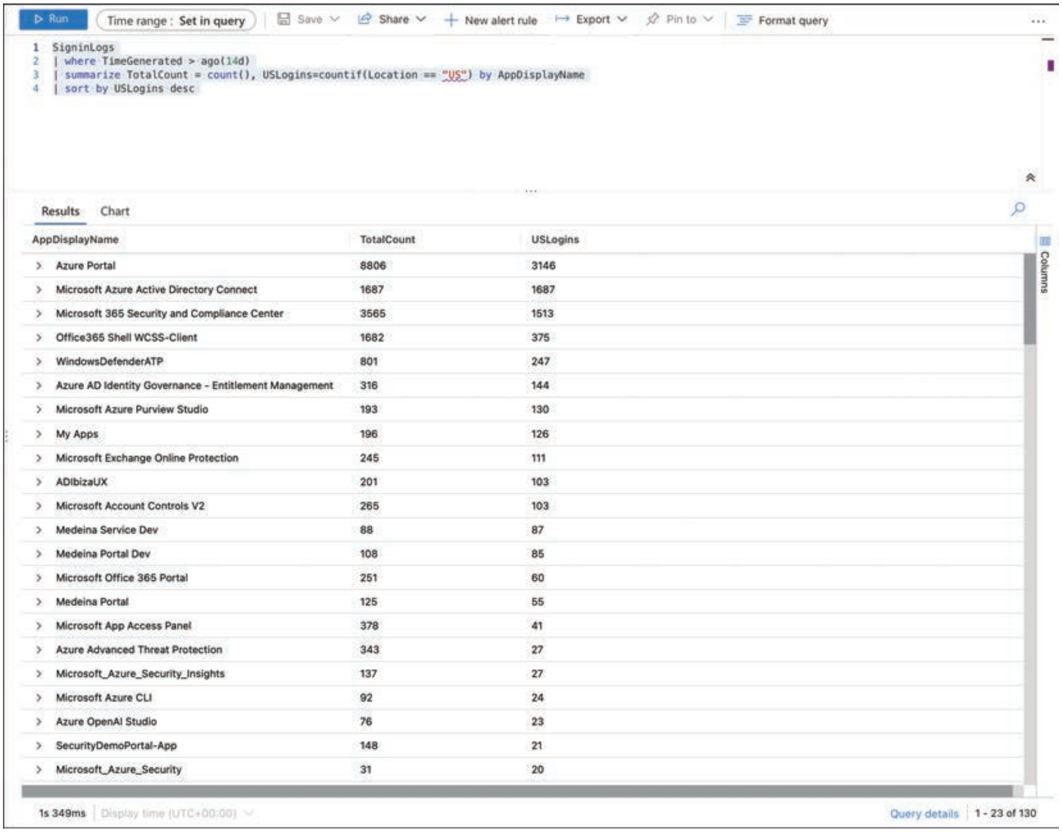


FIGURE 2-12 Total logins per application and total US logins

This view is much easier to read than two separate queries. Those with a sharp eye will also notice that we combined two `summarize` aggregate functions. Like how we combined multiple data-filtering methods in Chapter 1, we can do some powerful things by combining those functions. We highlight a few of those throughout this chapter.

Tip These queries can help you get a sense of what is happening with your devices in Intune. The first query will show you the count of successful create, delete, and patch events for the last seven days. The second will show the number of device enrollment successes and failures broken out by operating system. Looking for patterns and changes can help indicate something is not working as expected. –Mark Hopper, Senior Product Manager

```
IntuneAuditLogs
| where TimeGenerated > ago(7d)
| where ResultType == "Success"
| where OperationName has_any ("Create", "Delete", "Patch")
| summarize Operations=count() by OperationName, Identity
| sort by Operations, Identity
```

```
IntuneOperationalLogs
| where OperationName == "Enrollment"
| extend PropertiesJson = todynamic(Properties)
| extend OS = tostring(PropertiesJson["Os"])
| extend EnrollmentTimeUTC = todatetime(PropertiesJson["EnrollmentTimeUTC"])
| extend EnrollmentType = tostring(PropertiesJson["EnrollmentType"])
| project OS, Date = format_datetime(EnrollmentTimeUTC, 'M-d-yyyy'), Result
| summarize
    iOS_Successful_Enrollments = countif(Result == "Success" and OS == "iOS"),
    iOS_Failed_Enrollments = countif(Result == "Fail" and OS == "iOS"),
    Android_Successful_Enrollmenst = countif(Result == "Success" and
OS == "Android"),
    Android_Failed_Enrollments = countif(Result == "Fail" and OS == "Android"),
    Windows_Succesful_Enrollments = countif(Result == "Success" and
OS == "Windows"),
    Windows_Failed_Enrollments = countif(Result == "Fail" and OS == "Windows")
    by Date
```

Going a step further, how many unique user agents are using that application in that US region? Again, we could run separate queries like before, but combining them is much more useful, so we will use the `dcountif()` to only count the distinct rows that evaluate to `true` based on the expression. Run the following query; the output should be similar to Figure 2-13:

```
SigninLogs
| where TimeGenerated > ago(14d)
| summarize TotalCount = count(), USUserAgent=dcountif(UserAgent,
Location == "US") by AppDisplayName
| where USUserAgent > 0
| sort by USUserAgent desc
```

The `dcountif` function evaluates the column you want to have the distinct count of when the expression is evaluated to `true`. In this example, we are looking for the unique number of user agents when the location is `US`. Next, we grouped them by application display name (`AppDisplayName`).

You'll also notice we then have another `where` operator after `summarize`. So far in this book, we have filtered first and then done something with the output. You can continue filtering your query to drill down to the data you are interested in. In this example, we then filter out all the results that don't have a value and sort by descending order so the largest is at the top. Filtering and re-analyzing the data will be something we do repeatedly in the more advanced chapters of the book.

There is one last thing to know about `dcount()` and `dcountif()`. Earlier, we said that it provides an estimate of distinct values. If you need complete accuracy, you can use `count_distinct()` or `count_distinctif()`, which are limited to 100 million unique values. We are trading accuracy for speed because `dcount()` and `dcountif()` functions estimate based on the cardinality of the dataset. They are also less resource-intensive. If you only need an estimate, use `dcount()` or `dcountif()`.

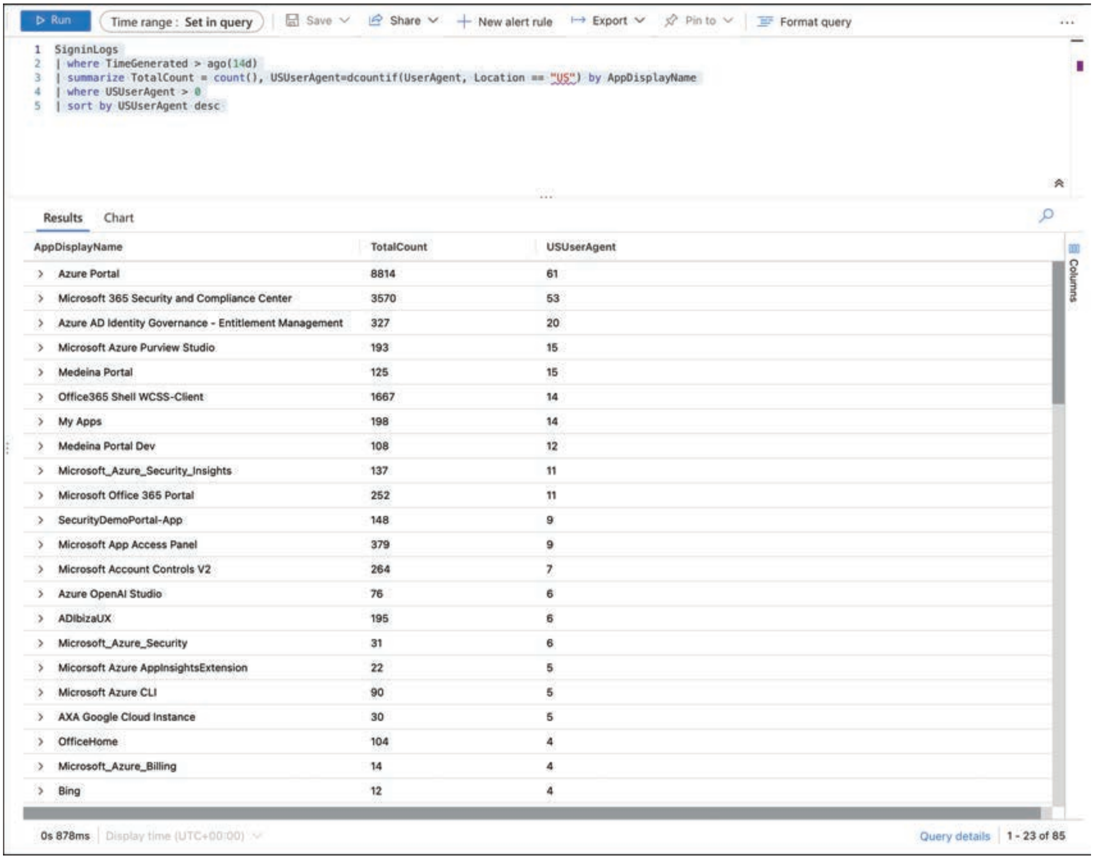


FIGURE 2-13 Total logins per application and by US access

Min, Max, Average, and Sum

Counting totals and determining the distinct number of rows is just the start when it comes to using `summarize`. There are many additional statistical types of information we'll frequently want to pull from our dataset, such as determining the first and last time something occurred. Perhaps you want to determine the average number of connections to a resource or the total amount of disk space consumed by your resources. There are aggregate functions to help you calculate these quickly.

Determining the Min and Max

A common scenario that will come up more often than you think is determining the first or last occurrence of something. You can use the `min()` or `max()` functions to find the minimum or maximum value of what is passed to it, such as finding the first time someone signed in to an application. Run the following query; your output should be similar to Figure 2-14:

```
SigninLogs
| where TimeGenerated > ago (14d)
| summarize TotalCount = count(), FirstEvent=min(TimeGenerated) by AppDisplayName
| sort by FirstEvent asc
```

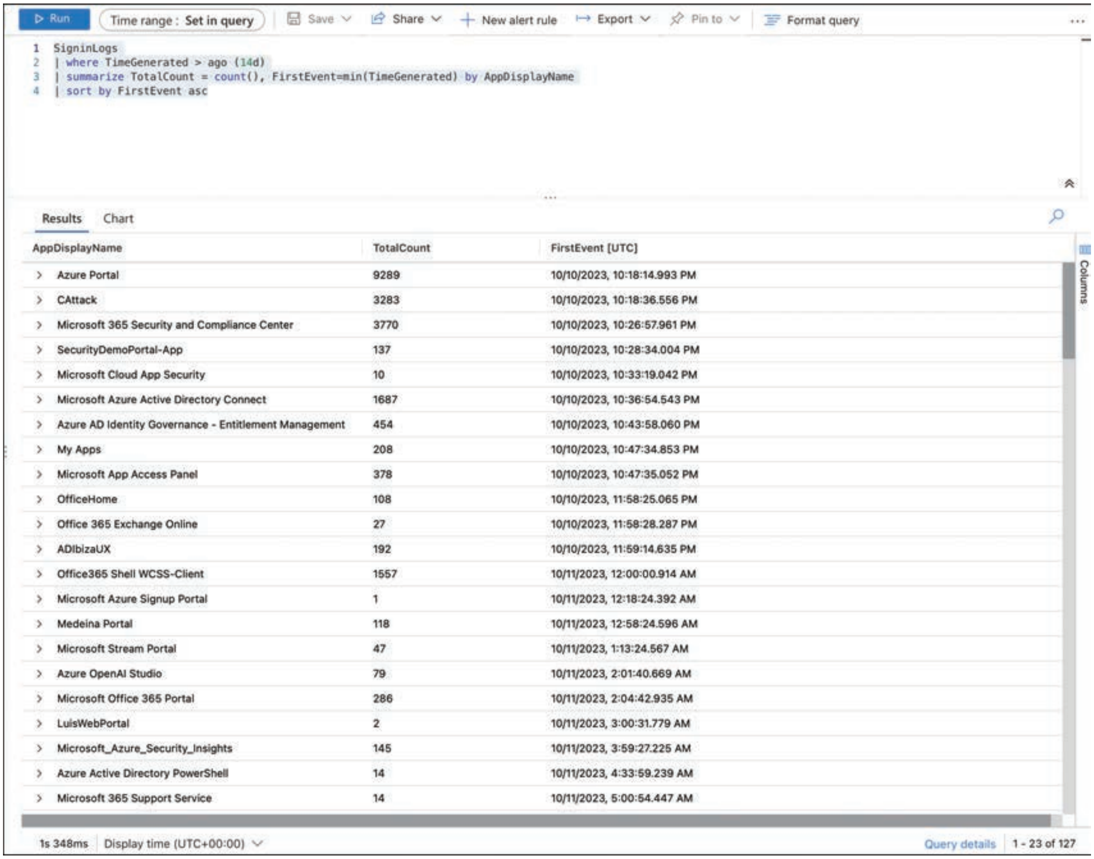
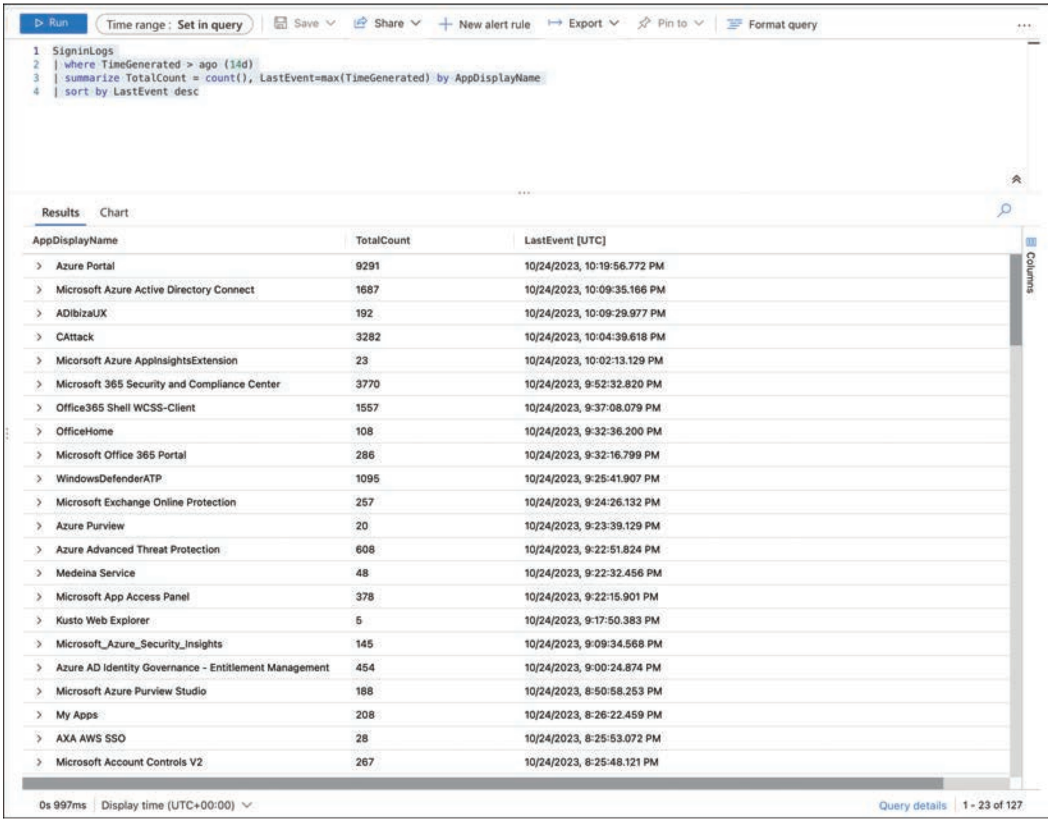


FIGURE 2-14 The first sign-in event in the application and the total sign-ins for that app

We can now quickly determine the first time a sign-in event was generated for that application and sort our results based on the earliest time. We can also do the opposite and determine the last time a sign-in event occurred for an application. To do that, we'll use the `max` function. Update the query to match the one listed here; the output should be similar to Figure 2-15.

```
SigninLogs
| where TimeGenerated > ago (14d)
| summarize TotalCount = count(), LastEvent=max(TimeGenerated) by AppDisplayName
| sort by LastEvent desc
```



AppDisplayName	TotalCount	LastEvent [UTC]
Azure Portal	9291	10/24/2023, 10:19:56.772 PM
Microsoft Azure Active Directory Connect	1687	10/24/2023, 10:09:35.166 PM
ADIbizaUX	192	10/24/2023, 10:09:29.977 PM
CAttack	3282	10/24/2023, 10:04:39.618 PM
Micorsoft Azure AppInsightsExtension	23	10/24/2023, 10:02:13.129 PM
Microsoft 365 Security and Compliance Center	3770	10/24/2023, 9:52:32.820 PM
Office365 Shell WCSS-Client	1557	10/24/2023, 9:37:08.079 PM
OfficeHome	108	10/24/2023, 9:32:36.200 PM
Microsoft Office 365 Portal	286	10/24/2023, 9:32:16.799 PM
WindowsDefenderATP	1095	10/24/2023, 9:25:41.907 PM
Microsoft Exchange Online Protection	257	10/24/2023, 9:24:26.132 PM
Azure Purview	20	10/24/2023, 9:23:39.129 PM
Azure Advanced Threat Protection	608	10/24/2023, 9:22:51.824 PM
Medeina Service	48	10/24/2023, 9:22:32.456 PM
Microsoft App Access Panel	378	10/24/2023, 9:22:15.901 PM
Kusto Web Explorer	5	10/24/2023, 9:17:50.383 PM
Microsoft_Azure_Security_Insights	145	10/24/2023, 9:09:34.568 PM
Azure AD Identity Governance - Entitlement Management	454	10/24/2023, 9:00:24.874 PM
Microsoft Azure Purview Studio	188	10/24/2023, 8:50:58.253 PM
My Apps	208	10/24/2023, 8:26:22.459 PM
AXA AWS SSO	28	10/24/2023, 8:25:53.072 PM
Microsoft Account Controls V2	267	10/24/2023, 8:25:48.121 PM

FIGURE 2-15 The last sign-in event in the application and the total sign-ins for the app

The output is similar to our last result but now shows the last sign-in event for that application. As mentioned earlier, we can combine multiple `summarize` functions to refine our results further. We can get a side-by-side timeline view of the first and last events with just the `min` and `max` functions. Run the following query; your results should be similar to the output in Figure 2-16:

```
SigninLogs
| where TimeGenerated > ago (14d)
| summarize TotalCount = count(), FirstEvent = min(TimeGenerated),
LastEvent=max(TimeGenerated) by AppDisplayName
| project AppDisplayName, TotalCount, FirstEvent, LastEvent
| sort by FirstEvent asc, LastEvent desc
```

Here, we are combining a few things that we've used so far in this book:

1. First, we use our new `min` and `max` aggregate functions to easily pull out the first and the last time a sign-in event occurred.
2. Next, we re-order the column's output to put the functions' results side by side to make it easier to see the difference.
3. Finally, we sort both columns, starting with the first event and then the last.

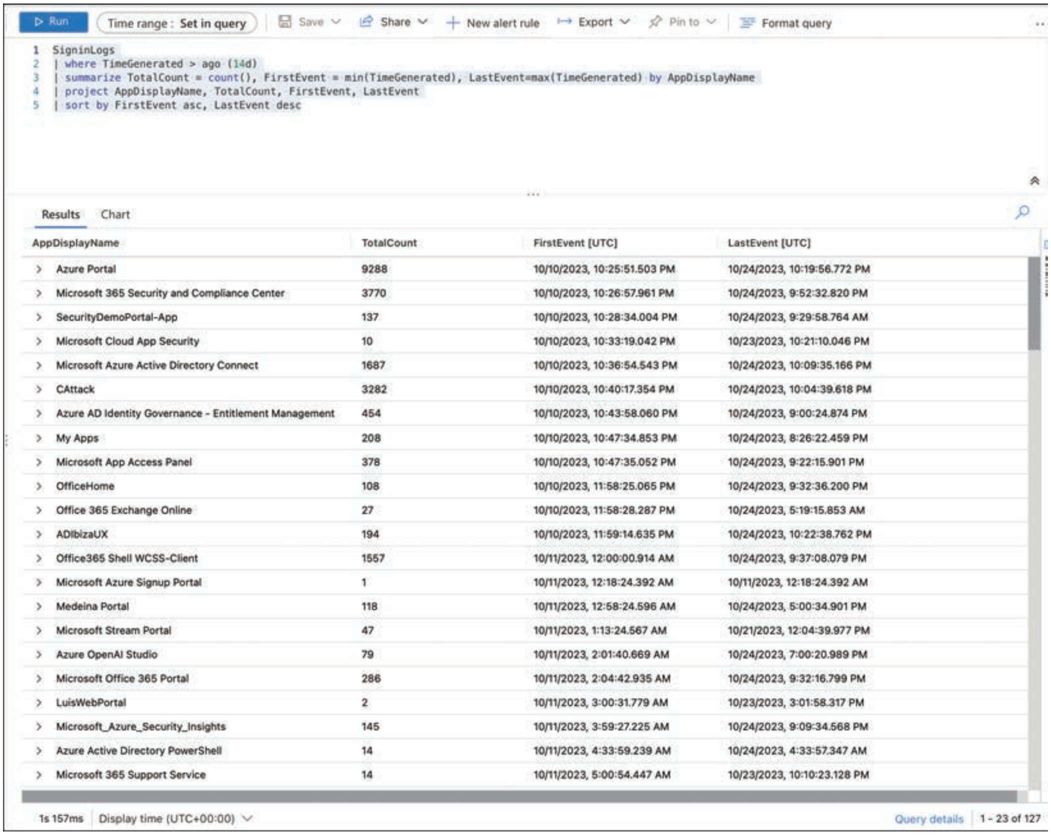


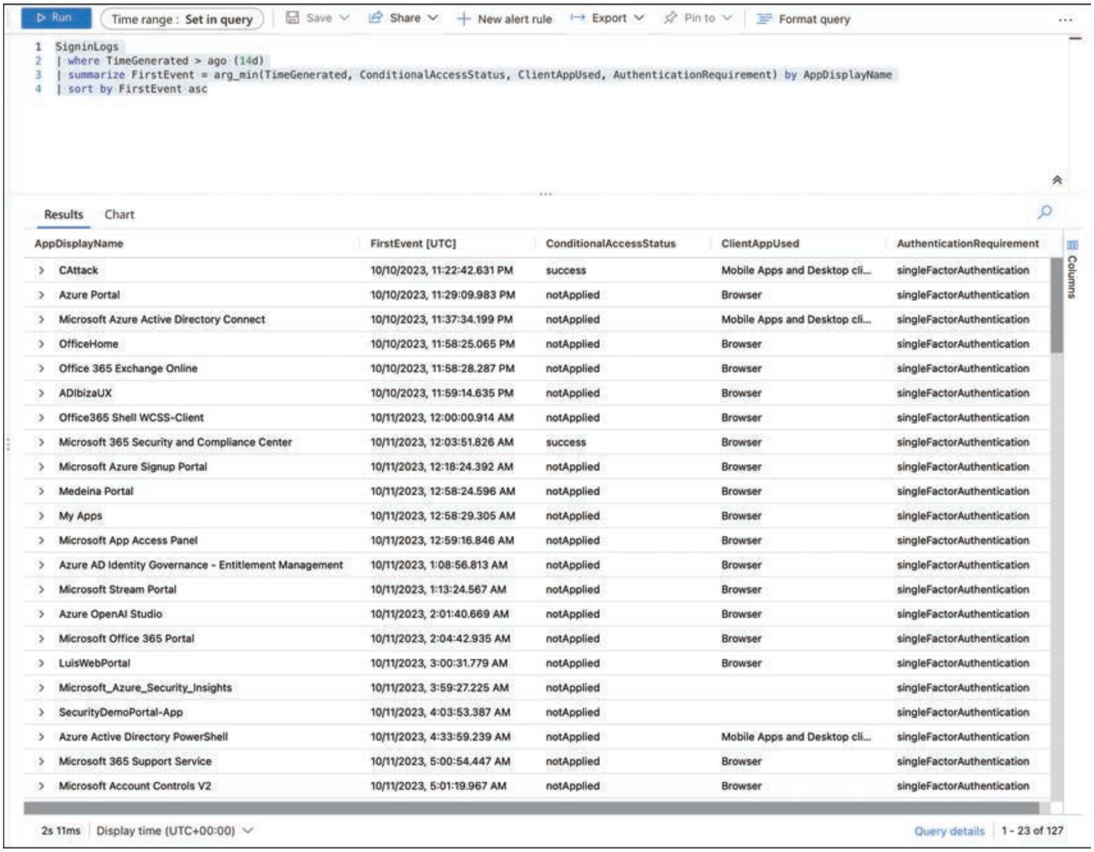
FIGURE 2-16 The first and last sign-in event for each application and the total sign-ins for each application

As we move into more advanced queries, you will see this similar pattern of combining multiple functions and filters, continuing to refine the query, and then formatting the output. You could easily add a filter for a specific user account to see this same information but for that user account.

Both `min()` and `max()` functions have a corresponding `minif()` and `maxif()` function. These work similarly to the `countif()` and `dcountif()` functions, where you can provide an expression to be evaluated; if the expression evaluates to `true`, it will then determine their `min` and `max` range.

The `min` and `max` functions return the value of a column, but what if you want the values for additional columns or find the columns where that value is located? You would use the `arg_min()` and `arg_max()` aggregate functions. You would provide the first column for which you want to find the minimum or maximum values, followed by the other columns for which you'd also like these values returned. You'd enter an asterisk (*) for all columns. Run the following query to find the minimum values of `TimeGenerated`; your output will be similar to Figure 2-17:

```
SigninLogs
| where TimeGenerated > ago (14d)
| summarize FirstEvent = arg_min(TimeGenerated, ConditionalAccessStatus,
ClientAppUsed, AuthenticationRequirement) by AppDisplayName
| sort by FirstEvent asc
```



AppDisplayName	FirstEvent [UTC]	ConditionalAccessStatus	ClientAppUsed	AuthenticationRequirement
CAttack	10/10/2023, 11:22:42.631 PM	success	Mobile Apps and Desktop cli...	singleFactorAuthentication
Azure Portal	10/10/2023, 11:29:09.983 PM	notApplied	Browser	singleFactorAuthentication
Microsoft Azure Active Directory Connect	10/10/2023, 11:37:34.199 PM	notApplied	Mobile Apps and Desktop cli...	singleFactorAuthentication
OfficeHome	10/10/2023, 11:58:25.065 PM	notApplied	Browser	singleFactorAuthentication
Office 365 Exchange Online	10/10/2023, 11:58:28.287 PM	notApplied	Browser	singleFactorAuthentication
ADIbizaUX	10/10/2023, 11:59:14.635 PM	notApplied	Browser	singleFactorAuthentication
Office365 Shell WCSS-Client	10/11/2023, 12:00:00.914 AM	notApplied	Browser	singleFactorAuthentication
Microsoft 365 Security and Compliance Center	10/11/2023, 12:03:51.826 AM	success	Browser	singleFactorAuthentication
Microsoft Azure Signup Portal	10/11/2023, 12:18:24.392 AM	notApplied	Browser	singleFactorAuthentication
Medeina Portal	10/11/2023, 12:58:24.596 AM	notApplied	Browser	singleFactorAuthentication
My Apps	10/11/2023, 12:58:29.305 AM	notApplied	Browser	singleFactorAuthentication
Microsoft App Access Panel	10/11/2023, 12:59:16.846 AM	notApplied	Browser	singleFactorAuthentication
Azure AD Identity Governance - Entitlement Management	10/11/2023, 1:08:56.813 AM	notApplied	Browser	singleFactorAuthentication
Microsoft Stream Portal	10/11/2023, 1:13:24.567 AM	notApplied	Browser	singleFactorAuthentication
Azure OpenAI Studio	10/11/2023, 2:01:40.669 AM	notApplied	Browser	singleFactorAuthentication
Microsoft Office 365 Portal	10/11/2023, 2:04:42.935 AM	notApplied	Browser	singleFactorAuthentication
LuisWebPortal	10/11/2023, 3:00:31.779 AM	notApplied	Browser	singleFactorAuthentication
Microsoft_Azure_Security_Insights	10/11/2023, 3:59:27.225 AM	notApplied		singleFactorAuthentication
SecurityDemoPortal-App	10/11/2023, 4:03:53.387 AM	notApplied		singleFactorAuthentication
Azure Active Directory PowerShell	10/11/2023, 4:33:59.239 AM	notApplied	Mobile Apps and Desktop cli...	singleFactorAuthentication
Microsoft 365 Support Service	10/11/2023, 5:00:54.447 AM	notApplied	Browser	singleFactorAuthentication
Microsoft Account Controls V2	10/11/2023, 5:01:19.967 AM	notApplied	Browser	singleFactorAuthentication

FIGURE 2-17 The minimum value of TimeGenerated by application with the additional columns specified

Here, we are looking for the minimum value of `TimeGenerated`—the first result showing an application sign-in event. Then, we also included additional columns we want to see the values of when `TimeGenerated` is at its minimum value, such as conditional access status, the client application used to access the application, and finally, whether it was a single-factor or multifactor request. We can run a similar query using the `arg_max` and return all columns using a `*`. Run the following query; your output will be similar to Figure 2-18:

```
SigninLogs
| where TimeGenerated > ago (14d)
| summarize LastEvent = arg_max(TimeGenerated, *) by AppDisplayName
| sort by LastEvent desc
```

This is similar to the minimum-value results, except we start with the most recent event and return all the columns in the table. The scrollbar at the bottom of Figure 2-18 shows that we have many more output columns to see all the values for each application's most recent event.

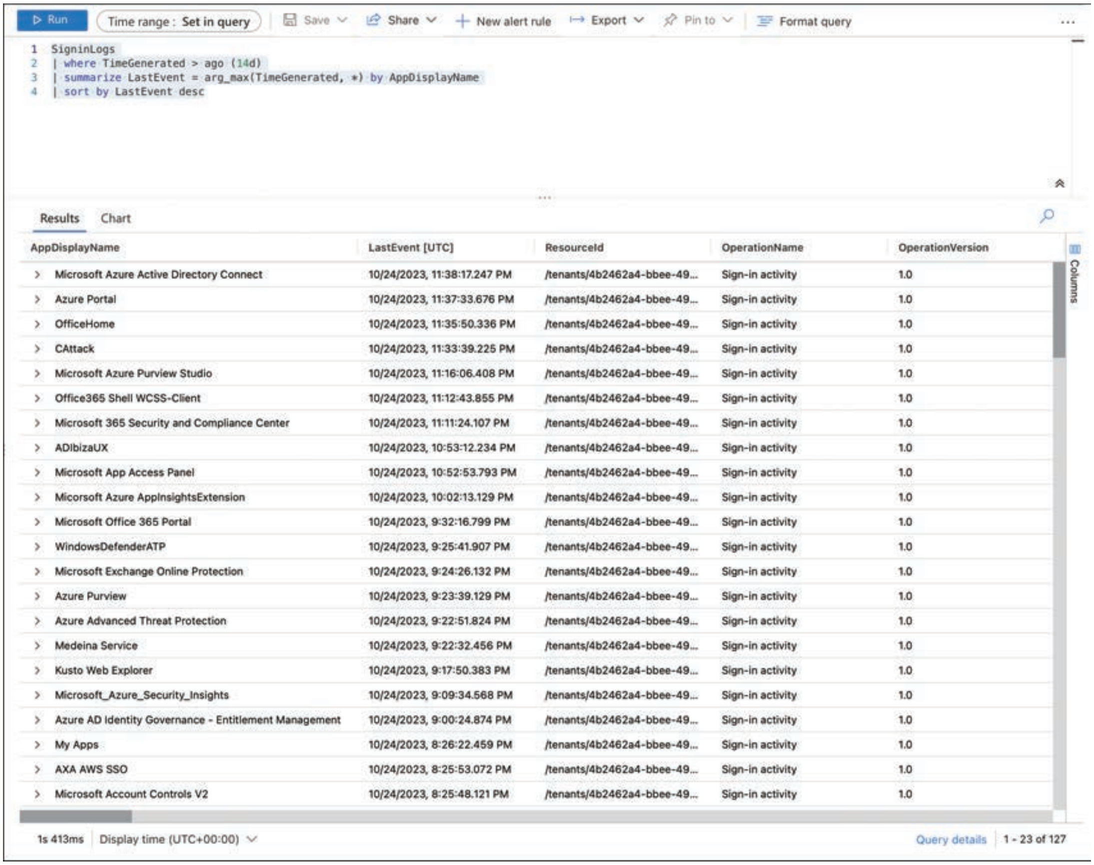


FIGURE 2-18 Maximum value

Determining the Average and Sum

The final set of statistical functions we'll look at in this section are `average` and `summation`. Just as you learned in school, these functions will find the `avg()`, otherwise known as the arithmetic mean, and `sum()`, which will find the sum of values in a column. Let's run the following query to understand how these work; your output should be similar to Figure 2-19:

```
SigninLogs
| where TimeGenerated > ago (14d)
| summarize AvgCreatedTime = avg(CreatedDateTime)by AppDisplayName
```

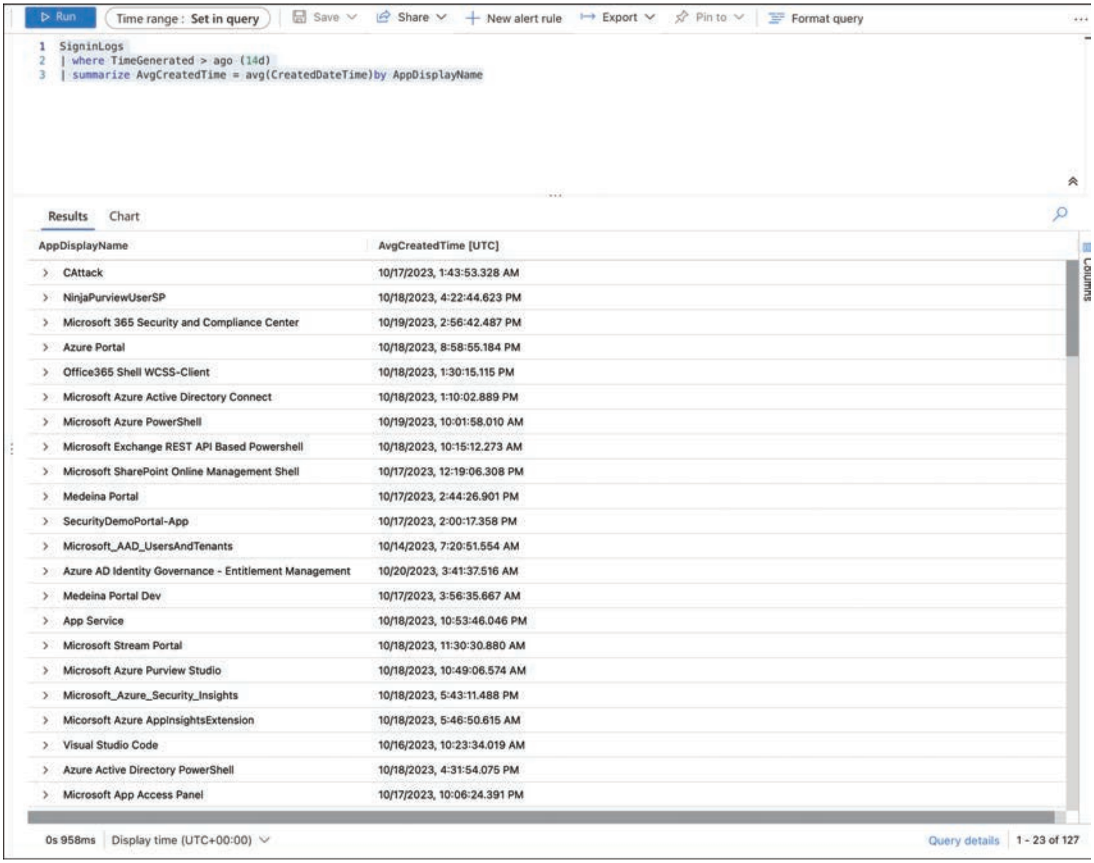


FIGURE 2-19 The average time when a sign-in event occurred for each application

Here, we can see the average time an event was created per application. We can also expand this with the `avgif()` function. Like our previous aggregate functions that use an `if` function, we can evaluate an expression; if its results are `true`, that expression is used for the calculation. For this, let's determine the average creation date if the user signed in from the US. Run the following query; your results should be similar to Figure 2-20:

```
SigninLogs
| where TimeGenerated > ago (14d)
| summarize AvgCreatedTime = avgif(CreatedDateTime, Location == "US")by
AppDisplayName
```

Similar to our previous results, we are now filtering on the average creation time if the sign-in came from the US. Some good examples of when to use average would be calculating the processor utilization or memory consumption of our IaaS virtual machines or even more advanced functionality from our Internet of Things (IoT) devices that might be reporting the temperature and humidity of their locations.

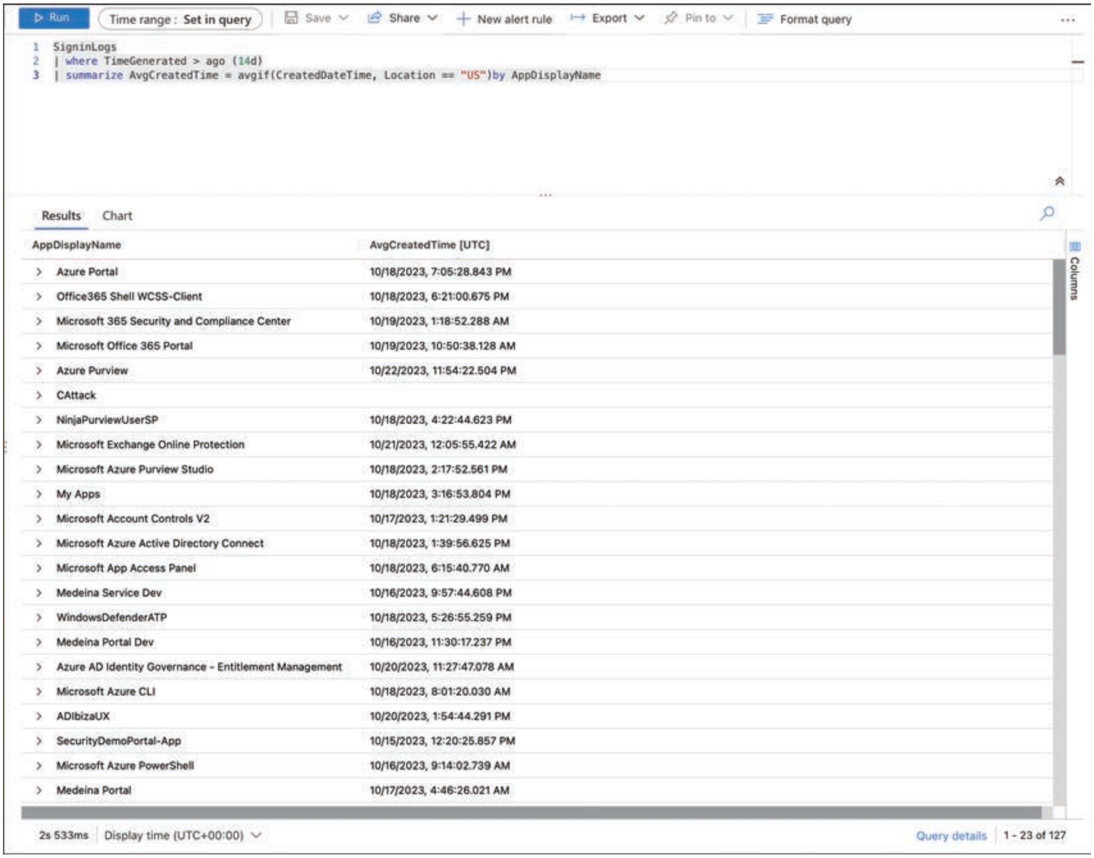


FIGURE 2-20 Average time when a US sign-in occurred for each application

> **Tip** This query looks at common performance metrics for virtual machines to help you look at resource consumption and if the virtual machines are sized correctly. –Laura Hutchcroft, Senior Service Engineer

```
Perf
| where TimeGenerated > ago(1h)
| where (ObjectName == "Processor" and CounterName == "% Processor Time") or
        (ObjectName == "Memory" and CounterName == "Available MBytes")
| summarize avg(CounterValue) by Computer, CounterName
```

The next aggregate functions we will look at are `sum()` and `sumif()`. For these, you simply provide the column you want to summarize. The data type value in the column needs to be numeric, such as a decimal, double, long, or integer. For more information on data types, see Chapter 1, "Data Types and Statements." Our sample sign-in logs don't have any good columns to sum, so we are using a different table, `AppPerformanceCounters`, for this query because it has more data with values that can be totaled. Run the following query; the results should be similar to Figure 2-21:

```
AppPerformanceCounters
| where  TimeGenerated > ago(14d)
| summarize sum(Value) by AppRoleName, Name
```

AppRoleName	Name	sum_Value
Web	% Processor Time	500722.11094931635
Web	Private Bytes	3386529271808
Web	Available Bytes	253702247841792
Web	Requests/Sec	2113.4763452616294
Web	Request Execution Time	8410889.280171633
ch1-usagegenfuncy37ha6	% Processor Time	1217.4080794844047
ch1-usagegenfuncy37ha6	Private Bytes	6918930444288
ch1-usagegenfuncy37ha6	% Processor Time Normalized	608.7040397422023
ch1-usagegenfuncy37ha6	IO Data Bytes/sec	33490970.372881357
ch1-usagegenfuncy37ha6	Requests/Sec	5507.0847457627115
ch1-usagegenfuncy37ha6	Request Execution Time	34728
ch1-usagegenfuncy37ha6	# of Exceps Thrown / sec	2014.3389830508472
ch1-usagegenfuncy37ha6	Requests In Application Queue	0
CH1-JavaWebApp	% Processor Time	390254.12336868444
CH1-JavaWebApp	Private Bytes	8498098512552
CH1-JavaWebApp	% Processor Time Normalized	11278.968178622834
CH1-JavaWebApp	Available Bytes	51932299206656
CH1-JavaWebApp	IO Data Bytes/sec	224382057.1035867
Fabrikam-App	% Processor Time	271274.98362579383
Fabrikam-App	Private Bytes	5700552085504
Fabrikam-App	% Processor Time Normalized	8297.532555584796
Fabrikam-App	Available Bytes	107226544140288
Fabrikam-App	IO Data Bytes/sec	122531391.97973633
Fabrikam-App	Requests/Sec	7835.979571893813
Fabrikam-App	Request Execution Time	4721606
Fabrikam-App	# of Exceps Thrown / sec	25971.611397629604

FIGURE 2-21 The sum of the application performance counters

Going through these performance counters for an application is a bit outside of the scope of this book, but the aggregate functions used so far can be applied to this table and columns. Understanding how much time an application has been executing or how much memory it has consumed might highlight places for optimization to drive some of the consumption costs down.

We can see that the Fabrikam-App handles 7,835 requests per second, more than ch1-usagegenfuncy37ha6, which performs 5,507 requests per second. We could have made this easier to read by only displaying that column. See "Visualizing Data" later in this chapter to see how to graph this data.

So far, everything we've been looking at is just doing the aggregate function for the 14-day `timespan` we've provided. In the previous example, Fabrikam-App handled 7,835 requests per second over those 14 days. Was one day busier for that application than another? Which day was the slowest day? Can we reduce our resource count? You could change your query to be only for the last day and run it daily, or you can have KQL do that using a concept called *binning*, which is covered next.

Bins, Percentages, and Percentiles

As we continue to analyze more of our data, we'll often need ways to group this data out by different segments to answer questions. What day of the week was the most active? Which month of the year was the least active? We will use a common technique called binning to accomplish this and more. We'll also frequently need to quickly convert the data into something a little easier to understand. Showing the percentage and the 25th or 95th percentile distribution for the data will help you tell a story with the data.

Grouping Data By Values (Binning)

Binning, or as you'll see it called, the `bin()` or `floor()` function, allows you to group your datasets by a smaller, specific set of values. The `bin` function takes two parameters:

- The first is the value you want to round down. This can be the `int`, `long`, `real`, `datetime`, or `timespan` types. (You'll end up using `timespan` often.)
- The second parameter is the bin size by which the values will be divided. This can be the `int`, `long`, `real`, or `timespan` types.

The most common type of binning will be by a date interval, frequently using a per-day interval. The `bin` function would be `bin(TimeGenerated, 1d)`. Another type of binning could be on different size groupings. For example, you could query how much free space was on a disk for your entire fleet and then bin them by intervals of 50 GB to see how many fall into each bucket.

Let’s run through a few examples of using per-day bins. Run the following query; your results should be similar to Figure 2-22.

```
SigninLogs
| where  TimeGenerated > ago(14d)
| where ResultType == 0
| summarize SuccessfullSignIn=count() by bin(TimeGenerated, 1d)
| sort  by  TimeGenerated asc
```

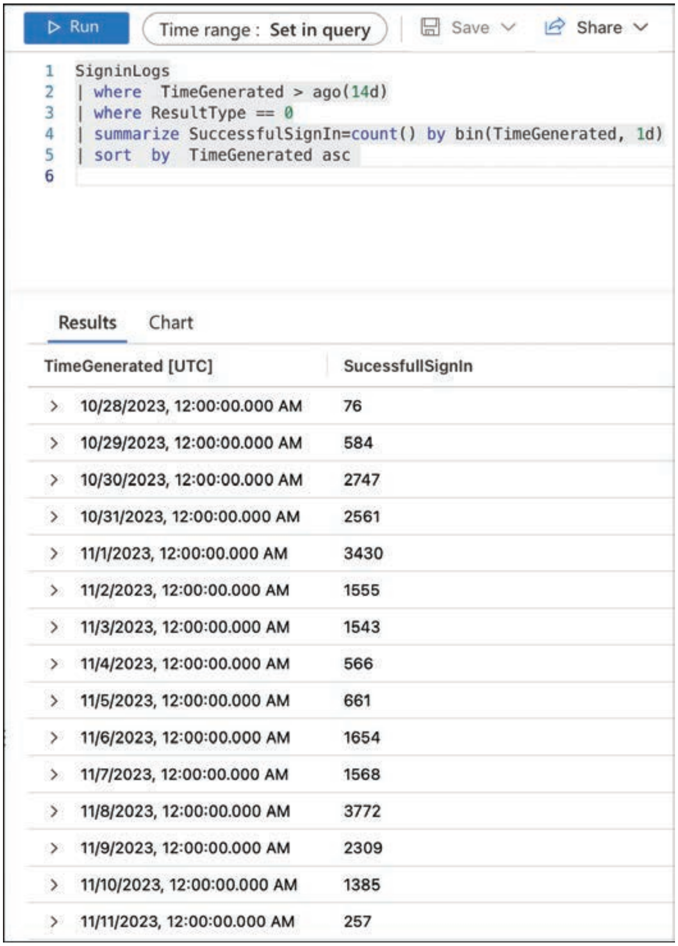


FIGURE 2-22 Daily Successful sign-in count

We are first filtering for how successful sign-ins are. In the previous examples, we counted them for those 14 days, but now you can see some days are busier than most. For most organizations, this is expected as people are off not working on the weekend. But the ability to bin by date is extremely useful. We’ll use this functionality multiple times throughout this book.

Let's also look at our previous application example, where we looked at how many requests per second it performed. We can simply add a binning technique to our existing query to break that summarized column by that daily time interval. Run the following query; your output should be similar to Figure 2-23:

```
AppPerformanceCounters
| where  TimeGenerated > ago(14d)
| where Name == "Requests/Sec" and AppRoleName == "Fabrikam-App"
| summarize sum(Value) by AppRoleName, Name, bin (TimeGenerated, 1d)
| project TimeGenerated, AppRoleName, Name, sum_Value
| sort by TimeGenerated asc
```

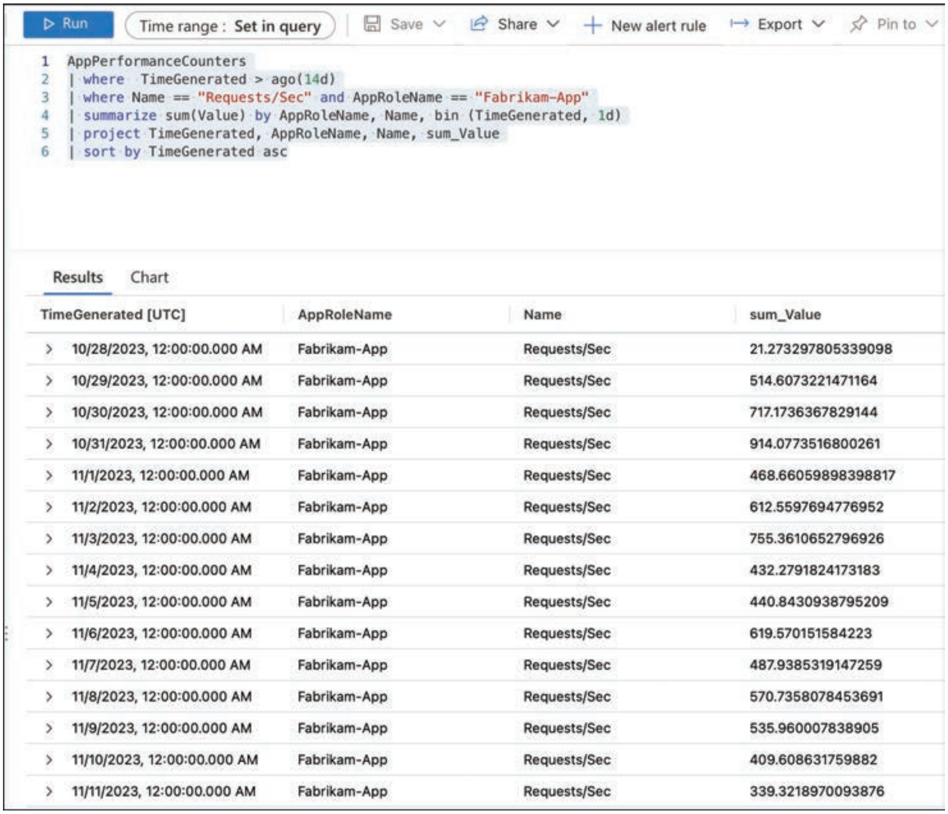


FIGURE 2-23 Total requests per second, per day

We made a few small modifications to the original query. First, we only filtered for the application and performance counter we were interested in. Our `summarize` function is the same as before, except we added a 1-day `bin` interval. We then cleaned up the output and sorted by date. If you wished any of the previous queries had been broken down by different intervals, feel free to alter them using the `bin` function!

> **Tip** This query looks at network flows per hour for the last 24 hours. Look for patterns and suspicious or long-running network flows. See *https://aka.ms/KQLMSPress/NetFlows* for set-up requirements. –Laura Hutchcroft, Senior Service Engineer

```
AzureNetworkAnalytics_CL
| where TimeGenerated > ago(24h)
| summarize sum(InboundFlows_d), sum(OutboundFlows_d) by bin(TimeGenerated, 1h)
```

Percentage

Calculating percentages is another common task. There is no built-in "to percentage" function, but we can calculate things using the `todouble()` function, dividing values, and multiplying results by 100—just as you would by hand. Let's use an example with real-life recommendations and combine it with some of the new KQL skills you've picked up so far. What is the percentage of sign-ins using single-factor authentication versus multifactor authentication? The `summarize count()` functions will tally the number of each authentication method, and then we use extend to calculate the percentage. Run the following query; your results should be similar to Figure 2-24:

```
SigninLogs
| where TimeGenerated > ago (14d)
| where ResultType == 0
| project TimeGenerated, AppDisplayName, UserPrincipalName, ResultType, ResultDes
cription,AuthenticationRequirement, Location
| summarize TotalCount=count(),MultiFactor=countif(AuthenticationRequirement ==
"multiFactorAuthentication"), SingleFactor=countif(AuthenticationRequirement ==
"singleFactorAuthentication")
| extend ['MFA Percentage']=(todouble(MultiFactor) * 100 / todouble(TotalCount))
| extend ['SFA Percentage']=(todouble(SingleFactor) * 100 / todouble(TotalCount))
```

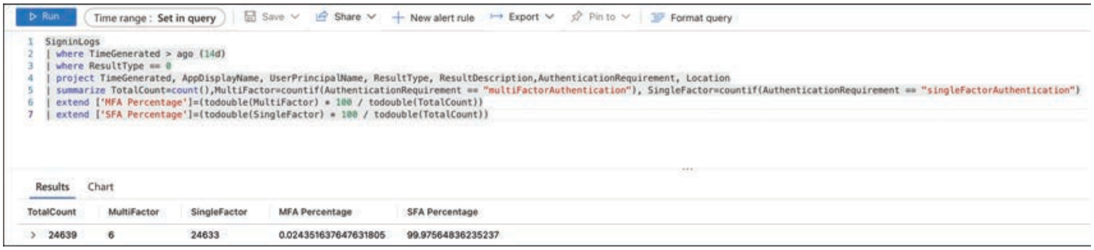


FIGURE 2-24 Percentage of MFA and single-factor sign-ins

Thankfully, this is a test environment because those numbers look bad. If you see similar numbers in your production environment, stop reading and roll out multifactor authentication immediately.

Let's break down this query. The beginning is the normal stuff, where we filter by time and successful sign-ins. Then, we pull the columns we want to work with and summarize the total count of all sign-ins, and then totals depending if the sign-ins are single-factor or multifactor.

Now, we will calculate the percentage of single-factor and multifactor by taking each integer total and casting the single-factor count and mulitfactor count to double using the todouble() function and multiplying by 100. Remember, as covered in the "Numerical Operators" section in Chapter 1, the data types can impact your results for numerical calculations. As you can see below, we have less than 1 percent of multifactor authentication sign-ins!

We can also round these results using the `round()` function, where you pass in the number you want to round and how much precision you want. We'll use `2` and `3` digits in the query below to show you the difference. Update your previous query to the following; your results will be similar to Figure 2-25:

```
SigninLogs
| where TimeGenerated > ago (14d)
| where ResultType == 0
| project TimeGenerated, AppDisplayName, UserPrincipalName, ResultType, ResultDes
cription,AuthenticationRequirement, Location
| summarize TotalCount=count(),MultiFactor=countif(AuthenticationRequirement ==
"multiFactorAuthentication"), SingleFactor=countif(AuthenticationRequirement ==
"singleFactorAuthentication")
| extend ['MFA Percentage']=round((todouble(MultiFactor) * 100 /
todouble(TotalCount)), 2)
| extend ['SFA Percentage']=round((todouble(SingleFactor) * 100 /
todouble(TotalCount)), 3)
```

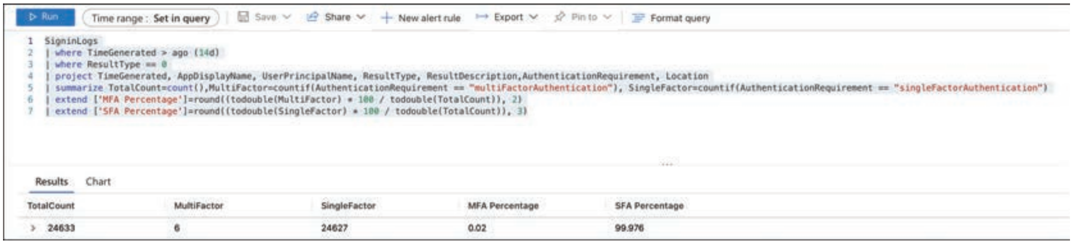


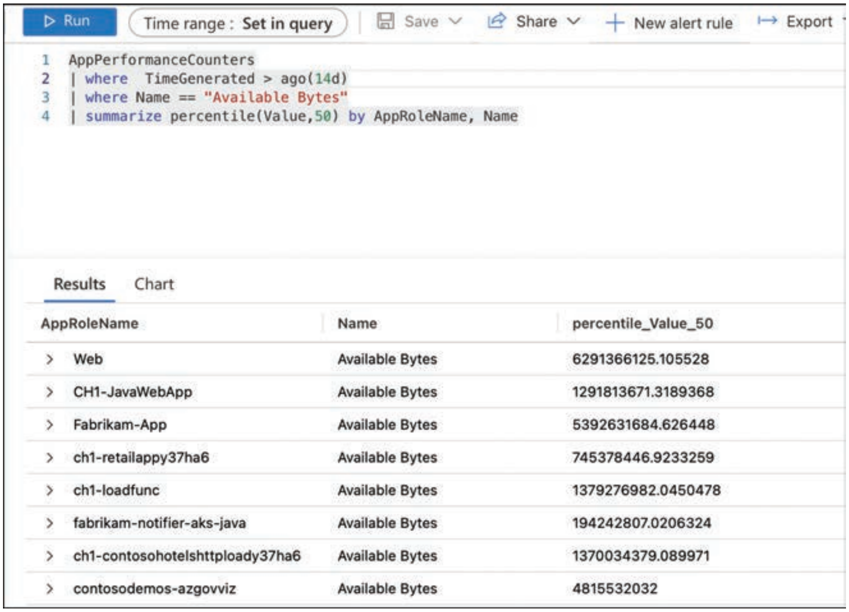
FIGURE 2-25 The rounded percentage of multifactor sign-ins and single-factor sign-ins

As you can see, you can round and alter how many digits you want to round to. This will be one of those common tactics you use repeatedly to calculate the percentage.

Percentiles

What if you wanted to determine if the values for the column are larger than a specific percentage compared to the other data? For that, we'll need to use the `percentile()` or `percentiles()` functions. `Percentile()` takes two parameters: the column you want to use for the calculation, and then the percentage you want to determine is equal to or larger than for that sample set. `Percentiles()` works similarly, except you can specify multiple comma-separated values. Let's go back to the `Application-PerformanceCounters` table and run the following query; your results should be similar to Figure 2-26:

```
AppPerformanceCounters
| where  TimeGenerated > ago(14d)
| where Name == "Available Bytes"
| summarize percentile(Value,50) by AppRoleName, Name
```

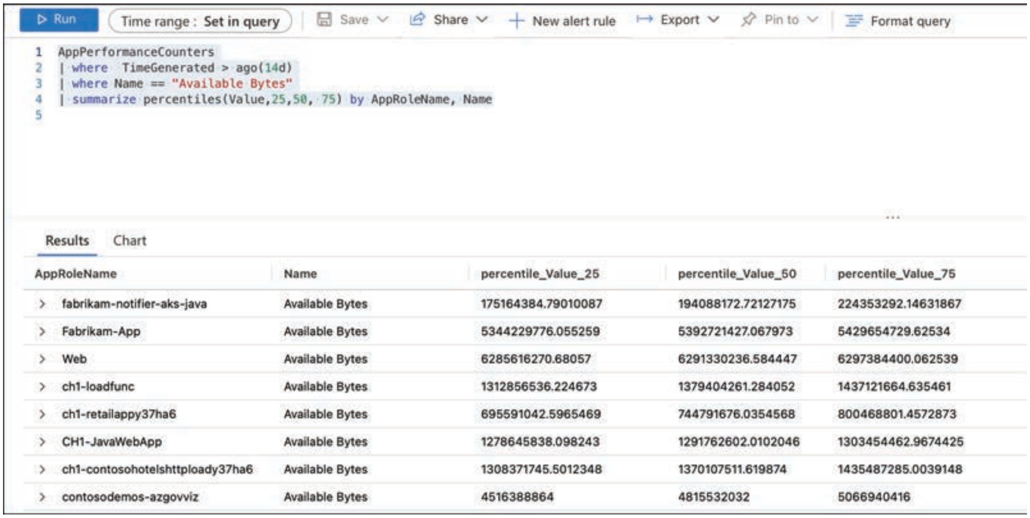


AppRoleName	Name	percentile_Value_50
Web	Available Bytes	6291366125.105528
CH1-JavaWebApp	Available Bytes	1291813671.3189368
Fabrikam-App	Available Bytes	5392631684.626448
ch1-retailappy37ha6	Available Bytes	745378446.9233259
ch1-loadfunc	Available Bytes	1379276982.0450478
fabrikam-notifier-aks-java	Available Bytes	194242807.0206324
ch1-contosohotelshttploady37ha6	Available Bytes	1370034379.089971
contosodemos-azgovviz	Available Bytes	4815532032

FIGURE 2-26 The 50th percentile value for Available Bytes per application

Here, we can see the value of Available Bytes that would be 50 percent or larger of the values for each application. We can get the values for multiple percentages using percentiles(). Update your command to the following; your output will be similar to Figure 2-27:

```
AppPerformanceCounters
| where  TimeGenerated > ago(14d)
| where Name == "Available Bytes"
| summarize percentiles(Value,25,50, 75) by AppRoleName, Name
```



AppRoleName	Name	percentile_Value_25	percentile_Value_50	percentile_Value_75
fabrikam-notifier-aks-java	Available Bytes	175164384.79010087	194088172.72127175	224353292.14631867
Fabrikam-App	Available Bytes	5344229776.055259	5392721427.067973	5429654729.62534
Web	Available Bytes	6285616270.68057	6291330236.584447	6297384400.062539
ch1-loadfunc	Available Bytes	1312856536.224673	1379404261.284052	1437121664.635461
ch1-retailappy37ha6	Available Bytes	695591042.5965469	744791676.0354568	800468801.4572873
CH1-JavaWebApp	Available Bytes	1278645838.098243	1291762602.0102046	1303454462.9674425
ch1-contosohotelshttploady37ha6	Available Bytes	1308371745.5012348	1370107511.619874	1435487285.0039148
contosodemos-azgovviz	Available Bytes	4516388864	4815532032	5066940416

FIGURE 2-27 The 25th, 50th, and 75th percentile values for available bytes per application

These values fall along the 25 percent, 50 percent, and 75 percent percentiles. This type of query is very interesting when you are trying to determine how to allocate and size resources such as virtual machine size or Azure App Service plan to pick for capacity planning or looking at usage spikes. You can also leverage this when looking for anomalies or outliers in your datasets. For example, if you have a simple test application that authenticates 100 times a day, that isn't the most concerning. However, if you looked at the percentiles of sign-ins and found that it was in the 95 percent percentile, that would probably be a big cause for concern. The simple test application should not be one of our environment's most logged-in applications. Either something is misconfigured, or it's being used in a way outside its normal scope. Percentiles can help highlight those types of behaviors.

Lists and Sets

We've been returning lots of interesting data so far in our KQL journey. What if we needed to temporarily store it to do some additional processing? For example, let's say when we returned all the `UserAgent` strings, we wanted to check them against a known set of known malicious user agents. Another scenario would be a compromised user account, and we want to be able to quickly determine all the unique applications they have accessed from the time of known compromise until we regained control of the account.

To be able to temporarily store some of these results or even create our own dataset, we'll use a common programming concept called a dynamic array. We'll cover more details of leveraging arrays in Chapter 3, "Advanced KQL Operators," and Chapter 5, "Security and Threat Hunting," but we'll use two very common functions—lists and sets—to get you started.

Lists

A list is pretty simple. You'll add items to the list either manually or as part of a `summarize` query. Let's first create our own list manually. Again, we'll cover this more in Chapter 5, "Security and Threat Hunting." Here, we're just looking at a simple example to get you started. Run the following query; your output will be similar to Figure 2-28:

```
let worldSeriesChampions = datatable (teamName: string, yearWon: int)
[
    "New York Yankees", 2000,
    "Arizona Diamondback", 2001,
    "Anaheim Angels", 2002,
    "Florida Marlins", 2003,
    "Boston Red Sox", 2004,
    "Chicago White Sox", 2005,
    "St. Louis Cardinals", 2006,
    "Boston Red Sox", 2007,
    "Philadelphia Phillies", 2008,
    "New York Yankees", 2009,
    "San Francisco Giants", 2010,
    "St. Louis Cardinals", 2011,
    "San Francisco Giants", 2012,
```

```
    "Boston Red Sox", 2013,
    "San Francisco Giants", 2014,
    "Kansas City Royals", 2015
];
worldSeriesChampions
| summarize mylist = make_list(teamName)
```

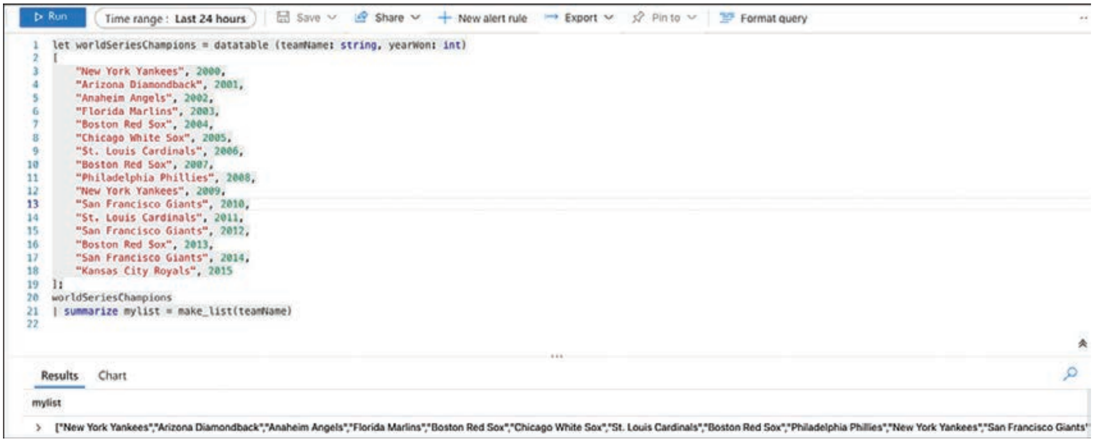


FIGURE 2-28 MLB World Series winners 2000–2015

Here, we can see the values—World Series winners from 2000 to 2015—inputted into this list. The New York Yankees and St. Louis Cardinals appear twice in the output. The list will store whatever is inputted, including multiple values of the same thing. But you can now manipulate this data as we've done throughout this chapter. Let's group these winners by even and odd years. Update your query; the output should be similar to Figure 2-29.

```
let worldSeriesChampions = datatable (teamName: string, yearWon: int)
[
    "New York Yankees", 2000,
    "Arizona Diamondback", 2001,
    "Anaheim Angels", 2002,
    "Florida Marlins", 2003,
    "Boston Red Sox", 2004,
    "Chicago White Sox", 2005,
    "St. Louis Cardinals", 2006,
    "Boston Red Sox", 2007,
    "Philadelphia Phillies", 2008,
    "New York Yankees", 2009,
    "San Francisco Giants", 2010,
    "St. Louis Cardinals", 2011,
    "San Francisco Giants", 2012,
    "Boston Red Sox", 2013,
    "San Francisco Giants", 2014,
    "Kansas City Royals", 2015
];
worldSeriesChampions
| summarize mylist = make_list(teamName) by isEvenYear= yearWon % 2 == 0
```

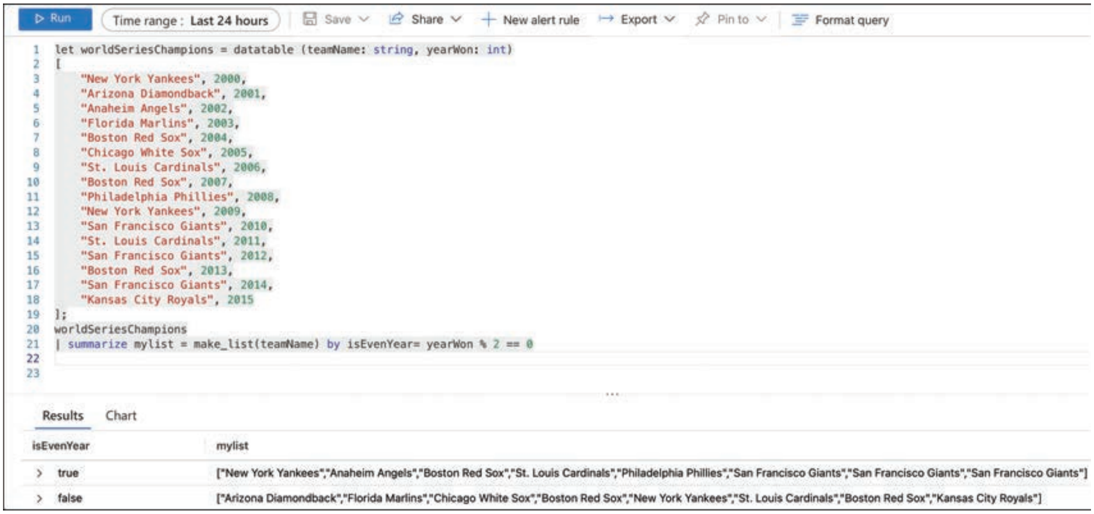


FIGURE 2-29 MLB World Series winners 2000–2015, by even- or odd-numbered years

The San Francisco Giants sure seem to do well in even-numbered years. This data is just for fun but demonstrates you can input your own dataset and perform different aggregate techniques. Let's go back to our built-in sample data and use a different function to make a list—the `make_list_if()` function. This will work similarly to the previous `if` functions we've seen throughout this chapter, where an expression evaluated as `true` will be added to the list. Run the following query; your output will be similar to Figure 2-30:

```
SigninLogs
| where TimeGenerated > ago (14d)
| summarize RiskLevels= make_list_if(RiskEventTypes_V2, RiskState == "atRisk") by
AppDisplayName
```

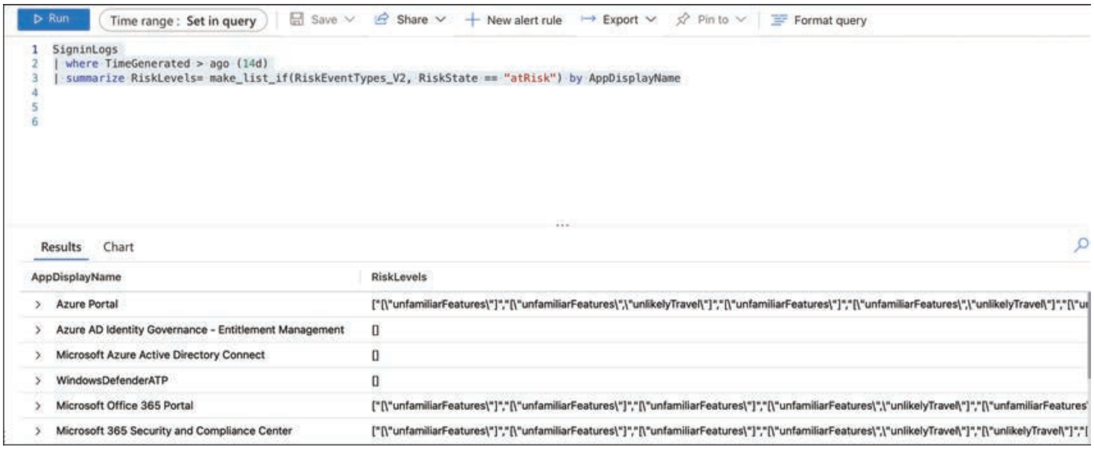


FIGURE 2-30 Applications with associated sign-in risk events

If the RiskState of a sign-in had risk indicated by the atRisk value, we then added the RiskEventType to the list. We then summarized this by application. In the output, we can see Azure Portal, Microsoft Office 365 Portal, and Microsoft 365 Security and Compliance Center have risky signs taking place. The other apps did not, so no risk events were added to their lists, essentially null lists. Depending on what you are trying to determine, you might want to remove the duplicate values. In other words, you might want only to store the distinct values. For that, we'll need to use sets.

Sets

The make_set() function works very similarly to the make_list, except it only stores the distinct values. Let's rerun our previous World Series champions query, but instead of making a list, let's make a set. The output should be similar to Figure 2-31.

```
let worldSeriesChampions = datatable (teamName: string, yearWon: int)
[
    "New York Yankees", 2000,
    "Arizona Diamondback", 2001,
    "Anaheim Angels", 2002,
    "Florida Marlins", 2003,
    "Boston Red Sox", 2004,
    "Chicago White Sox", 2005,
    "St. Louis Cardinals", 2006,
    "Boston Red Sox", 2007,
    "Philadelphia Phillies", 2008,
    "New York Yankees", 2009,
    "San Francisco Giants", 2010,
    "St. Louis Cardinals", 2011,
    "San Francisco Giants", 2012,
    "Boston Red Sox", 2013,
    "San Francisco Giants", 2014,
    "Kansas City Royals", 2015
];
worldSeriesChampions
| summarize myset = make_set(teamName) by isEvenYear= yearWon % 2 == 0
```

Notice that each team only appears once in that set, whereas previously, the San Francisco Giants appeared multiple times in the even-year list. This is because only distinct values are stored.

The make_set_if() function works similarly to make_list_if(), but once again, it will only store distinct values. Let's rerun our previous make_list_if() query but store it as a set instead. The output should be similar to Figure 2-32:

```
SigninLogs
| where TimeGenerated > ago (14d)
| summarize RiskLevels= make_set_if(RiskEventTypes_V2, RiskState == "atRisk") by
AppDisplayName
```

Index

B

C

D

E

F

G

H

I

M

N

O

P

T

U

V

W

X-Y-Z