

advanced

MUST LEARN

KQL

THIS IS THE WAY

ROD TRENT

Welcome, data adventurers, to *Must Learn KQL: Advanced Edition!* Whether you're a seasoned query crafter or a curious analyst ready to level up, this book is your guide to mastering the Kusto Query Language (KQL) with confidence and flair. Building on the foundations of KQL, this advanced edition dives deep into the powerful, nuanced features of KQL, designed to transform you into a data wizard capable of taming even the wildest datasets in Azure Data Explorer and beyond.

In these pages, you'll embark on a journey through complex queries, time-series analysis, and performance optimization, all while uncovering KQL's hidden gems. Inspired by the creative spirit of past explorations – like Captain KQL's superhero saga or Archmage Kusto's epic battles against data dragons – this book blends practical expertise with engaging analogies to make advanced concepts accessible and memorable. Expect hands-on examples, real-world scenarios, and challenges that push your skills to new heights, all crafted to spark your curiosity and creativity.

Drawing from the vibrant community around resources like the *Must Learn KQL Workshop* (check out the modules at <https://github.com/rod-trent/MustLearnKQL/tree/main/Workshop>), this book is tailored for those ready to wield KQL's full potential. Whether you're analyzing Azure logs, hunting for insights in massive datasets, or building custom solutions, you'll find tools and techniques to make your queries faster, sharper, and more impactful.

So, grab your virtual spellbook, fire up your KQL sandbox, and let's dive into the art and science of advanced data querying. Your next big data breakthrough awaits!

NOTE: In the original *Must Learn KQL* book, many examples relied on the *aka.ms/LADemo* environment for hands-on practice. However, that environment is no longer available (it still exists, but the awesome data that it once had is no longer there), marking the end of an era for early KQL learners. Fear not! *Must Learn KQL: Advanced Edition* adapts to this change by providing its own carefully curated datasets for examples and exercises, ensuring you have everything you need to master advanced KQL concepts. Alternatively, you can run the queries in this book using the free Kusto cluster at <https://aka.ms/kustofree>, a versatile and accessible platform for honing your skills. Whether you're exploring our provided datasets or leveraging the free cluster, you're set for a seamless and impactful learning experience.

BIGGER NOTE: I know it can be tough reading a book with KQL samples that you can't use right away or that require significant effort to implement and use. To make things easier, you can search my blog site (<https://rodtrent.com>) for the practical query examples included in this book to complement your learning. You can use each chapter title as the search string to locate the original queries.

BIGGEST NOTE: When I first released *Must Learn KQL* in August 2023, I could only hope it would resonate with those diving into Kusto Query Language. Little did I know the incredible journey it would spark – one fueled entirely by you, the curious minds, the late-night coders, and the teams tackling real-world data challenges.

Your passion has turned this book into something extraordinary. From the GitHub repository that started it all, you've showered it with 1,030 stars and 173 forks, creating a vibrant hub of contributions and discussions. The traffic speaks volumes too: over 222,000 GitHub views from 58,295 unique visitors, and 3,520,000 blog views from 95,000 unique visitors. It's humbling to see how you've shared, explored, and built upon these pages.

On the publishing front, your support has been nothing short of phenomenal. The eBook (PDF) has been downloaded 50,000 times, while the Kindle edition has sold over 1,600 copies and racked up over 17,000 pages read through Kindle Unlimited. In print, over 1,000 paperbacks and over 190 hardcovers have found their way into hands eager for knowledge. And the assessment? Over 12,000 completions and 7,000+ certificates delivered – proof that you're not just reading, but mastering KQL and applying it with confidence.

This success isn't mine; it's ours. Thank you for your reviews, your shares, your questions, and your triumphs. You've made *Must Learn KQL* more than a book – it's a community, a toolkit, and a launchpad for what's next.

That success has led to the birth of this *advanced* book. This book represents the culmination of everything I've worked on with

KQL since the Must Learn KQL series was first released, designed to help you master KQL with real-world applications.

Here's to querying smarter, together.

With deepest gratitude,

- Rod

Rod's Blog is a reader-supported publication. To receive new posts and support my work, consider becoming a free or paid subscriber.

Table of Contents

Benefits of the mv-expand Operator 12

Challenges and Potential Pitfalls 13

Best Practices for Using mv-expand 14

Examples of mv-expand Usage 15

Understanding the Join Operator 49

Types of Joins 50

Optimizing Performance with Joins 54

Practical Examples 55

Why JSON in KQL? 63

Getting Started with JSON in KQL 64

Handling Nested Structures 66

Optimizing Performance for Large Datasets 68

Flattening Arrays in JSON 69

Combining JSON Parsing with Aggregations 71

Best Practices for JSON Parsing in KQL 72

Overview of Regular Expressions in KQL	259
Applications in Data Cleaning	264
Key Considerations	267
Analyzing Trends in Malware Detections Over Time	302
Identifying Repeat Offenders or Recurring Attack Vectors	305
Strategic Implications	307
Understanding KQL	309
Anomaly Detection and Time-Series Analysis	310
Key Techniques in KQL for Time-Series Anomaly Detection	310
Working Examples	311
Visualization and Insights	315
Best Practices	315
Tracking Outdated Software and Configurations	320
Detecting Privilege Escalation and Improper Administrative Access	323
Next Steps	326
1. Revealing Anomalous Activities in Specific Geographic Locations or IP Ranges	328
2. Filtering Events Involving Known Malicious Entities	330

Best Practices for Using KQL in Threat Detection 332

Querying for Unusual Login Attempts or Failed Logins Across Devices 334

Identifying Patterns of Data Exfiltration or Unauthorized File Access 338

Best Practices for Filtering Logs 451

Best Practices for Aggregating Logs 453

Best Practices for Visualizing Logs 455

Advanced Tips 457

Understanding KQL and Its Strengths 501

Challenges of Querying Large Datasets 502

Strategies for Efficient KQL Querying 503

Advanced Techniques 506

Common Pitfalls to Avoid 508

The Fundamentals of KQL Datatypes 510

Casting and Conversion in KQL 512

Exploring KQL Operators 513

Practical Examples 515

Common Pitfalls and How to Avoid Them 516

Why Integrate KQL with Power Automate and PowerShell? 545

Integrating KQL with Power Automate 545

Integrating KQL with PowerShell 548

Combining Power Automate and PowerShell 551

The Prompt 663

Copilot 663

Grok 666

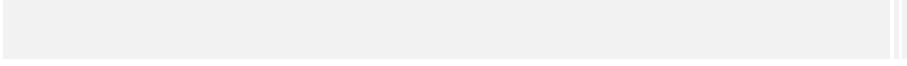
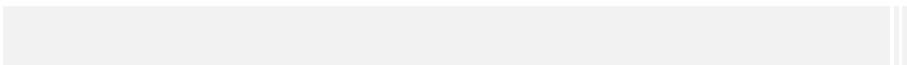
ChatGPT 677

Microsoft Security Copilot 685

Microsoft 365 Copilot 687

Gemini 693

Shameless Plug 739



Chapter 1

Understanding the KQL mv-expand Operator

KQL's mv-expand operator is designed to handle fields that contain arrays or multi-value data. When applied, it transforms each element in these multi-value fields into its own row. This operation is useful when individual analysis or manipulation of array elements is required.

For example, consider the following dataset:

- User: John
- Visited Pages: ["Home", "Products", "Contact"]
- User: Alice
- Visited Pages: ["About", "Careers"]

Using mv-expand on the Visited Pages field would result in:

- John, Home
- John, Products

- John, Contact
- Alice, About
- Alice, Careers

This transformation facilitates detailed analysis, such as determining the popularity of pages visited by users.

Benefits of the mv-expand Operator

1. Simplifies Data Exploration

When working with multi-value fields, mv-expand simplifies queries by presenting data in a tabular format that is more intuitive and easier to analyze.

2. Enables Element-specific Aggregations

By breaking down arrays into individual rows, users can perform aggregations or calculations specific to each element, such as counting occurrences or calculating averages.

3. Enhances Visualization Compatibility

Data expanded using mv-expand is often better suited for visualization tools that require a flat structure for charts and graphs.

Challenges and Potential Pitfalls

Despite its advantages, the mv-expand operator can lead to unexpected results if not used with proper understanding.

1. Data Duplication

Expanding multi-value fields may inadvertently duplicate data from other columns. For example, if a field contains user-related metadata alongside an array, expanding the array creates duplicate rows that repeat the metadata.

2. Performance Concerns

If applied to large datasets or high-cardinality multi-value fields, mv-expand can significantly

increase the number of rows, leading to performance issues.

3. Loss of Context

Depending on the structure of your data, expanding fields may obscure relationships between elements or lose contextual information crucial for the analysis.

Best Practices for Using mv-expand

1. Use Filters Before Expansion

To mitigate performance issues, filter your dataset to include only relevant rows before applying mv-expand. This reduces the size of the resulting table.

2. Combine with Summarize or Other Aggregations

After expanding the multi-value fields, consider applying aggregation operators like summarize to consolidate data and prevent row explosion.

3. Validate Results

Always inspect the output after using mv-expand. Verify that the transformation preserves meaningful relationships and doesn't yield unintended duplications.

4. Document Queries

Clearly document your queries that use mv-expand to ensure future users understand its purpose and behavior.

Examples of mv-expand Usage

Consider this example dataset:

- ID: 1
- Tags: ["Azure", "Cloud", "Data"]
- ID: 2
- Tags: ["AI", "Machine Learning"]
- ID: 3
- Tags: ["Big Data", "Analytics"]

Applying mv-expand to Tags will flatten the data:

- 1, Azure
- 1, Cloud

- 1, Data
- 2, AI
- 2, Machine Learning
- 3, Big Data
- 3, Analytics

This allows further analysis, such as counting tag occurrences:

```
datatable(ID: int, Tags: dynamic)
```

```
[
```

```
1, dynamic(["Azure", "Cloud", "Data"]),
```

```
2, dynamic(["AI", "Machine Learning"]),
```

```
3, dynamic(["Big Data", "Analytics"])
```

```
]
```

```
| mv-expand Tags
```

Result:

	ID	Tags
	1	Azure
	1	Cloud
	1	Data
	2	AI
	2	Machine Learning
	3	Big Data
	3	Analytics

The `mv-expand` operator in KQL is a powerful tool for working with multi-value fields, providing enhanced flexibility in data analysis. However, it requires careful handling to avoid issues like data duplication and row explosion. By following best practices and validating results, you can use `mv-expand` effectively to unlock deeper insights and improve decision-making in your data workflows.

Chapter 2

Understanding the KQL Parse Operator

The **Kusto Query Language (KQL)** is a powerful tool for querying large datasets in Azure Data Explorer, Azure Monitor, Microsoft Sentinel, and Microsoft Fabric. One of its versatile operators is the **parse operator**, which allows you to evaluate a string expression and parse its value into one or more calculated columns.

Syntax and Parameters

The basic syntax for the parse operator is:

```
T | parse [kind=kind [flags=regexFlags]]  
expression with [*] stringConstant columnName  
[:columnType] [*], ...
```

- **T**: The tabular input to parse.
- **kind**: Specifies the parsing mode (simple, regex, relaxed).
- **regexFlags**: Flags for regex mode (e.g., U for ungreedy, m for multi-line).

- **expression**: The string expression to evaluate.
- **stringConstant**: The string to search and parse.
- **columnName**: The name of the column to assign the parsed value.
- **columnType**: The type to convert the value to (default is string).

Supported Parsing Modes

1. **Simple**: Strict matching with regular string values.
2. **Regex**: Uses regular expressions for matching.
3. **Relaxed**: Allows partial matching, resulting in null values for unmatched types.

Practical Examples

Example 1: Parsing Developer Trace Statements

Consider a table with a column `EventText` containing strings like:

```
Event: NotifySliceRelease  
(resourceName=PipelineScheduler,  
totalSlices=27, sliceNumber=23,  
lockTime=02/17/2016 08:40:01,
```

```
releaseTime=02/17/2016 08:40:01,  
previousLockTime=02/17/2016 08:39:01)
```

Using the parse operator, you can extend the table with individual columns for each value:

```
Traces | parse EventText with * "resourceName=" resourceName ", totalSlices=" totalSlices: long  
* "sliceNumber=" sliceNumber: long *  
"lockTime=" lockTime ", releaseTime=" releaseTime: date "," * "previousLockTime=" previousLockTime: date ")" *
```

Example 2: Extracting Email Alias and DNS

For a table with contact information, you can parse email addresses and website URLs:

```
Leads | parse Contacts with * "email=" alias: string "@" domain: string ", Website=https:" WebsiteDomain: string ")" | project EmailAddress=strcat(alias, "@", domain), EmailAlias=alias, WebsiteDomain
```

Advanced Usage with Regex Flags

In regex mode, you can use flags to control the parsing behavior. For example, to handle newlines and case insensitivity:

```
Traces | parse kind=regex flags=Ui EventText  
with * "resourceName=" resourceName ',' *
```

The **parse operator** in KQL is a powerful tool for transforming and extracting data from string expressions. By understanding its syntax and modes, you can efficiently manipulate and analyze your data in Azure environments.

Chapter 3

Understanding Tabular Expression Statements in Kusto Query Language (KQL)

One of the core components of KQL is the **tabular expression statement**, which is essential for querying and manipulating tabular data.

What is a Tabular Expression Statement?

A tabular expression statement is typically what people refer to when they talk about queries in KQL. It is composed of **tabular data sources**, **tabular data operators**, and optional **rendering operators**. The statement follows a regular form represented by the pipe character (`|`), visually depicting the flow of data from left to right.

Syntax and Parameters

The syntax of a tabular expression statement is straightforward:

```
Source | Operator1 | Operator2 |  
RenderInstruction
```

- **Source**: A tabular data source, such as a table reference, a function invocation that returns a table, or a table literal.
- **Operator**: Tabular data operators like filters and projections.
- **RenderInstruction**: Optional rendering operators or instructions.

Tabular Data Sources

Tabular data sources produce sets of records that are processed by tabular data operators.

Supported sources include:

- **Table references**
- **Tabular range operator**
- **Print operator**
- **Function invocation returning a table**
- **Table literal ("datatable")**

Examples

Filtering Rows by Condition

This example counts the number of records in the `StormEvents` table where the `state` column has the value "FLORIDA":

```
StormEvents | where State == "FLORIDA" | count
```

Output:

Count

1042

Combining Data from Two Tables

In this example, the `join` operator combines records from the `StormEvents` table and the `PopulationData` table based on the `State` column:

```
StormEvents | where InjuriesDirect +  
InjuriesIndirect > 50 | join (PopulationData)  
on State | project State, Population,  
TotalInjuries = InjuriesDirect +  
InjuriesIndirect
```

Output:

State	Population	TotalInjuries
-------	------------	---------------

ALABAMA 4918690 60

CALIFORNIA 39562900 61

KANSAS 2915270 63

MISSOURI 6153230 422

OKLAHOMA 3973710 200

TENNESSEE 6886720 187

TEXAS 29363100 137

Tabular expression statements are a fundamental aspect of KQL, enabling users to efficiently query and manipulate tabular data. Understanding the syntax, parameters, and practical examples can significantly enhance your ability to work with KQL in various Microsoft services.

Chapter 4

Understanding the Restrict Statement in Kusto Query Language (KQL)

The restrict statement in Kusto Query Language (KQL) is a powerful tool for controlling access to data within a query. It limits the set of table or view entities visible to subsequent query statements, making it particularly useful for implementing row-level security in middle-tier applications. In this chapter, we'll explore the syntax, parameters, and practical examples of using the restrict statement to enhance data security and query management.

What is the Restrict Statement?

The restrict statement is used to limit access to specific entities within a database or cluster. By specifying which tables, views, or patterns are accessible, it ensures that only authorized data is visible to the rest of the query. This is especially useful for applications that need to enforce security policies or restrict user access to sensitive data.

Syntax and Parameters

The syntax for the restrict statement is straightforward:

```
restrict access to (EntitySpecifiers)
```

Parameters:

- **EntitySpecifiers**: A comma-separated list of entity specifiers, which can include:
 - Identifiers defined by a `let` statement as tabular views.
 - Table or function references similar to those used by a `union` statement.
 - Patterns defined by a pattern declaration.

Practical Examples

1. Limiting Access to a Specific Table

To restrict access to a single table within the current database:

```
restrict access to (database().Table1);
```

```
Table1 | count
```

2. Restricting Access Across Databases

To restrict access to tables in multiple databases:

```
restrict access to (database().Table1,  
database('DB2').Table2);
```

```
union (Table1), (database('DB2').Table2) |  
count
```

3. Using Let Statements

Define views using `let` statements and restrict access to these views:

```
let Test = () { print x = 1 };
```

```
restrict access to (Test);
```

```
Test
```

4. Applying Patterns

Use wildcard patterns to match multiple entities:

```
let Test1 = () { print x = 1 };
```

```
let Test2 = () { print y = 1 };
```

```
restrict access to (*); // Access is restricted  
to Test1, Test2, and no tables/functions are  
accessible.
```

5. Preventing User Access to Other User Data

A middle-tier application can prepend a user's query with a logical model to prevent access to other users' data:

```
let RestrictedData = view () { Data | where  
UserID == "username@domain.com" };
```

```
restrict access to (RestrictedData);
```

```
RestrictedData | summarize MonthlySalary =  
sum(Salary) by Year, Month
```

Use Cases

1. Row-Level Security

Implement row-level security by restricting access to views that filter data based on user identity:

```
let UserView = view () { UserData | where  
UserID == "current_user_id" };
```

```
restrict access to (UserView);
```

```
UserView | summarize TotalPurchases =  
sum(PurchaseAmount) by Month
```

2. Multi-Tenant Applications

Ensure tenants can only access their own data by using restrict statements to limit visibility:

```
let TenantData = view () { AllData | where  
TenantID == "tenant_id" };
```

```
restrict access to (TenantData);
```

```
TenantData | summarize TotalUsage =  
sum(UsageAmount) by Service
```

The **restrict** statement in KQL is a versatile tool for managing data access and enhancing security within queries. By understanding its syntax and practical applications, you can effectively control which entities are visible to your queries, ensuring that sensitive data remains protected and accessible only to authorized users. Whether you're implementing row-level security or managing multi-tenant applications, the **restrict** statement provides the flexibility and control needed to secure your data.

Chapter 5

Creating Geospatial Visualizations with Kusto Query Language (KQL)

Geospatial visualizations are a powerful way to analyze and interpret data based on geographical locations. Kusto Query Language (KQL) offers robust tools for performing geospatial clustering and visualizations, making it easier to understand spatial patterns and trends. In this chapter, we'll explore various methods for creating geospatial visualizations using KQL, including plotting points on a map, using GeoJSON values, and finding anomalies based on geospatial data.

Prerequisites

To run the queries in this tutorial, you need access to a query environment with sample data. You can use one of the following:

- A Microsoft account or Microsoft Entra user identity

- A Fabric workspace with Microsoft Fabric-enabled capacity

Plotting Points on a Map

To visualize points on a map, use the `project` operator to select the columns containing longitude and latitude. Then, use the `render` operator to display the results in a scatter chart with the `kind` set to `map`.

Example:

```
StormEvents | take 100 | project BeginLon,  
BeginLat | render scatterchart with (kind =  
map)
```

Plotting Multiple Series of Points

To visualize multiple series of points, select the longitude, latitude, and a third column that defines the series. This allows points to be colored differently based on their series.

Example:

```
StormEvents | take 100 | project BeginLon,  
BeginLat, EventType | render scatterchart with  
(kind = map)
```

Using GeoJSON Values to Plot Points

Dynamic GeoJSON values provide flexibility for real-time mapping applications.

Use `geo_point_to_s2cell` and `geo_s2cell_to_central_point` to map storm events in a scatter chart.

Example:

```
StormEvents | project BeginLon, BeginLat |  
summarize by hash =  
geo_point_to_s2cell(BeginLon, BeginLat, 5) |  
project point =  
geo_s2cell_to_central_point(hash) | project lng  
= toreal(point.coordinates[0]), lat =  
toreal(point.coordinates[1]) | render  
scatterchart with (kind = map)
```

Representing Data Points with Variable-Sized Bubbles

Visualize the distribution of data points by aggregating each cluster and plotting the central point with variable-sized bubbles.

Example:

```
StormEvents | where EventType == "Tornado" |  
project BeginLon, BeginLat | where  
isnotnull(BeginLat) and isnotnull(BeginLon) |  
summarize count_summary = count() by hash =  
geo_point_to_s2cell(BeginLon, BeginLat, 4) |  
project geo_s2cell_to_central_point(hash),  
count_summary | extend Events = "count" |  
render piechart with (kind = map)
```

Displaying Points Within a Specific Area

Define a region using a polygon and filter events within that region using `geo_point_in_polygon`.

Example:

```
let southern_california = dynamic({ "type":  
"Polygon", "coordinates": [[[[-119.5, 34.5], [-  
115.5, 34.5], [-115.5, 32.5], [-119.5, 32.5],  
[-119.5, 34.5]]]});
```

```
StormEvents | where  
geo_point_in_polygon(BeginLon, BeginLat,  
southern_california) | project BeginLon,  
BeginLat | summarize count_summary = count() by  
hash = geo_point_to_s2cell(BeginLon, BeginLat,
```

```
8) | project geo_s2cell_to_central_point(hash),  
count_summary | extend Events = "count" |  
render piechart with (kind = map)
```

Showing Nearby Points on a LineString

Find nearby storm events along a specified
LineString using `geo_distance_point_to_line`.

Example:

```
let roadToKeyWest = dynamic({ "type":  
  "linestring", "coordinates": [ [-  
    81.79595947265625, 24.56461038017685], [-  
    81.595458984375, 24.627044746156027], [-  
    81.52130126953125, 24.666986385216273], [-  
    81.35650634765625, 24.66449040712424], [-  
    81.32354736328125, 24.647017162630366], [-  
    80.8099365234375, 24.821639356846607], [-  
    80.62042236328125, 24.93127614538456], [-  
    80.37872314453125, 25.175116531621764], [-  
    80.42266845703124, 25.19251511519153], [-  
    80.4803466796875, 25.46063471847754]]});
```

```
StormEvents | where isnotempty(BeginLat) and  
isnotempty(BeginLon) | project BeginLon,  
BeginLat, EventType | where  
geo_distance_point_to_line(BeginLon, BeginLat,  
roadToKeyWest) < 500 | render scatterchart with  
(kind = map)
```

Showing Nearby Points in a Polygon

Find nearby storm events within a specified polygon using `geo_distance_point_to_polygon`.

Example:

```
let roadToKeyWest = dynamic({ "type": "polygon",  
  "coordinates": [[[ -80.08209228515625,  
    25.39117928167583], [-80.4913330078125,  
    25.517657429994035], [-80.57922363281249,  
    25.477992320574817], [-82.188720703125,  
    24.632038149596895], [-82.1942138671875,  
    24.53712939907993], [-82.13104248046875,  
    24.412140070651528], [-81.81243896484375,  
    24.43714786161562], [-80.58746337890625,  
    24.794214972389486], [-80.08209228515625,  
    25.39117928167583]]}));
```

```
StormEvents | where isnotempty(BeginLat) and  
isnotempty(BeginLon) | project BeginLon,  
BeginLat, EventType | where  
geo_distance_point_to_polygon(BeginLon,  
BeginLat, roadToKeyWest) < 500 | render  
scatterchart with (kind = map)
```

Finding Anomalies Based on Geospatial Data

Analyze storm events within a particular state using S2 cells and temporal aggregation to identify anomalies.

Example:

```
let stateOfInterest = "Texas";  
  
let statePolygon = materialize(US_States |  
    extend name =  
        tostring(features.properties.NAME) | where name  
        == stateOfInterest | project geometry =  
            features.geometry);  
  
let stateCoveringS2cells = statePolygon |  
    project s2Cells =  
        geo_polygon_to_s2cells(geometry, 9);  
  
StormEvents | extend s2Cell =  
    geo_point_to_s2cell(BeginLon, BeginLat, 9) |  
    where s2Cell in (stateCoveringS2cells) | where  
        geo_point_in_polygon(BeginLon, BeginLat,  
            toscalar(statePolygon)) | make-series damage =  
            avg(DamageProperty + DamageCrops) default =  
            double(0.0) on StartTime step 7d | extend  
            anomalies = series_decompose_anomalies(damage)
```

```
| render anomalychart with (anomalycolumns =  
anomalies)
```

Geospatial visualizations in KQL provide a powerful way to analyze and interpret data based on geographical locations. By leveraging operators like `project`, `render`, and geospatial functions such as `geo_point_to_s2cell`, you can create detailed and insightful visualizations. Whether you're plotting points on a map, analyzing clusters, or finding anomalies, KQL's geospatial capabilities offer the tools needed to unlock the full potential of your data.

Chapter 6

Mastering Aggregation Functions in Kusto Query Language (KQL)

Aggregation functions in Kusto Query Language (KQL) are essential for summarizing and analyzing large datasets. These functions allow you to group and combine data from multiple rows into meaningful summary values, such as counts, averages, and sums. In this chapter, we'll explore various aggregation functions and their applications, providing a comprehensive guide to mastering data aggregation in KQL.

Prerequisites

Before diving into aggregation functions, ensure you have access to a query environment with sample data, such as the `StormEvents` table. You can use a Microsoft account or Microsoft Entra user identity, and a Fabric workspace with Microsoft Fabric-enabled capacity.

Key Aggregation Functions

1. Summarize Operator

The `summarize` operator is fundamental for performing aggregations. It groups rows based on the `by` clause and uses specified aggregation functions to combine each group into a single row.

Example: Counting the number of events by state.

```
StormEvents | summarize TotalStorms = count()  
by State
```

2. Visualize Query Results

Visualizing query results helps identify patterns and trends. Use the `render` operator to display results in various chart formats.

Example: Displaying the previous query results in a bar chart.

```
StormEvents | summarize TotalStorms = count()  
by State | render barchart
```

3. Conditional Counting

Use `countif()` to count rows based on specific conditions, providing insights into data that meets certain criteria.

Example: Counting storms that caused crop damage.

```
StormEvents | summarize StormsWithCropDamage =  
countif(DamageCrops > 0) by State | top 5 by  
StormsWithCropDamage
```

4. Grouping Data into Bins

The `bin()` function groups data into numeric or time-based bins, aiding in understanding value distributions.

Example: Counting storms causing crop damage weekly in 2007.

```
StormEvents | where StartTime  
between(datetime(2007-01-01) .. datetime(2007-  
12-31)) and DamageCrops > 0 | summarize  
EventCount = count() by bin(StartTime, 7d) |  
render timechart
```

5. Calculating Min, Max, Avg, and Sum

Perform multiple aggregations in a single `summarize` operator to compute various summary values.

Example: Calculating crop damage statistics by event type.

```
StormEvents | where DamageCrops > 0 | summarize  
MaxCropDamage = max(DamageCrops), MinCropDamage =  
min(DamageCrops), AvgCropDamage =  
avg(DamageCrops) by EventType | sort by  
AvgCropDamage
```

Advanced Aggregation Techniques

1. Calculating Percentages

Understand data distribution by calculating percentages using `count()` and `countif()`.

Example: Percentage of storms causing crop damage by state.

```
StormEvents | summarize TotalStormsInState =  
count(), StormsWithCropDamage =  
countif(DamageCrops > 0) by State | extend  
PercentWithCropDamage =  
round((todouble(StormsWithCropDamage) /
```

```
TotalStormsInState * 100), 2) | sort by  
StormsWithCropDamage
```

2. Extracting Unique Values

Use `make_set()` to create arrays of unique values from table rows.

Example: Event types causing deaths by state.

```
StormEvents | where DeathsDirect > 0 or  
DeathsIndirect > 0 | summarize  
StormTypesWithDeaths = make_set(EventType) by  
State | project State, StormTypesWithDeaths |  
sort by array_length(StormTypesWithDeaths)
```

3. Bucketing Data by Condition

Group data into buckets based on specified conditions using the `case()` function.

Example: Grouping states by storm-related injuries.

```
StormEvents | summarize InjuriesCount =  
sum(InjuriesDirect) by State | extend  
InjuriesBucket = case(InjuriesCount > 50,  
"Large", InjuriesCount > 10, "Medium",  
InjuriesCount > 0, "Small", "No injuries") |
```

```
sort by State asc | summarize  
InjuryBucketByState = count() by InjuriesBucket  
| render piechart
```

4. Sliding Window Aggregations

Summarize columns using a sliding window to analyze data over time.

Example: Property damage analysis over a seven-day sliding window.

```
let windowStart = datetime(2007-07-01);
```

```
let windowEnd = windowStart + 13d;
```

```
StormEvents | where EventType in ("Tornado",  
"Flood", "Wildfire") | extend bin =  
bin_at(startofday(StartTime), 1d, windowStart)  
| extend endRange = iff(bin + 7d > windowEnd,  
windowEnd, iff(bin + 7d - 1d < windowStart,  
windowStart, iff(bin + 7d - 1d < bin, bin, bin  
+ 7d - 1d))) | extend range = range(bin,  
endRange, 1d) | mv-expand range to  
typeof(datetime) | summarize  
min(DamageProperty), max(DamageProperty),  
round(avg(DamageProperty)) by Timestamp =
```

```
bin_at(range, 1d, windowStart), EventType |  
where Timestamp >= windowStart + 7d;
```

Mastering aggregation functions in KQL empowers you to derive valuable insights from your data. By leveraging operators like `summarize`, `bin()`, and `countif()`, and advanced techniques like sliding window aggregations, you can efficiently analyze and visualize complex datasets. Whether you're exploring data trends or performing detailed analyses, KQL's aggregation functions provide the tools needed to unlock the full potential of your data.

Chapter 7

A Deep Dive into the KQL Join Operator

Kusto Query Language (KQL) is a sophisticated query language that allows users to extract insights from logs, telemetry, and data stored in Azure Monitor, Application Insights, and Log Analytics. The join operator in KQL is one of the most powerful tools for data correlation, enabling users to combine information from multiple datasets to uncover relationships and derive actionable insights.

In this chapter, we'll take a comprehensive look at the KQL join operator, exploring its syntax, use cases, types, and practical examples to help you master its capabilities.

Understanding the Join Operator

In KQL, the join operator is used to combine rows from two tables based on a common column or set of columns. The resulting dataset contains

data from both tables for matching rows, and it can be configured in several ways to suit various data analysis scenarios.

Basic Syntax

The basic syntax for a join operation is as follows:

```
Table1
```

```
| join kind=JoinType (Table2) on ColumnName
```

- **Table1:** The first dataset, known as the left table.
- **Table2:** The second dataset, known as the right table.
- **JoinType:** The type of join (e.g., inner, outer, leftouter).
- **ColumnName:** The column or columns used to match rows from both tables.

Types of Joins

KQL provides several types of joins to accommodate different analytical needs. Each type determines how rows from the two tables

are paired and what happens when there is no match in one or both tables.

1. Inner Join

The inner join selects only rows that have matching values in both tables. This is the default join type if no `kind` is specified.

Example:

```
Table1
```

```
| join (Table2) on CommonColumn
```

Use case: Identifying rows that exist in both datasets, such as matching customer IDs from two separate systems.

2. Left Outer Join

The left outer join includes all rows from the left table (Table1) and matches rows from the right table (Table2). If no match is found, columns from the right table will contain null values.

Example:

Table1

```
| join kind=leftouter (Table2) on CommonColumn
```

Use case: Retrieving all users from a user database and their corresponding purchases, even for those who made no purchases.

3. Right Outer Join

The right outer join is the opposite of the left outer join. It includes all rows from the right table and matches rows from the left table, with null values for unmatched rows.

Example:

Table1

```
| join kind=rightouter (Table2) on CommonColumn
```

Use case: Listing all purchase records and determining which users made each purchase, including those not present in the user database.

4. Full Outer Join

The full outer join includes all rows from both tables. For rows that do not have a match in the other table, the unmatched columns are filled with null values.

Example:

Table1

```
| join kind=fullouter (Table2) on CommonColumn
```

Use case: Combining error logs from two systems to identify overlapping and unique issues.

5. Anti Join

The anti join selects rows from one table that do not have a match in the other table.

Example:

Table1

```
| join kind=anti (Table2) on CommonColumn
```

Use case: Identifying users in a database who have not yet logged in or completed a task.

Optimizing Performance with Joins

While KQL joins offer immense flexibility, they can also be resource-intensive, particularly when working with large datasets. Here are some tips to optimize your join queries:

- **Filter Data Early:** Use where clauses on both tables to reduce the number of rows before performing the join.
- **Indexing:** Ensure that the columns used for joins are indexed for faster lookups.
- **Minimize Columns:** Select only the necessary columns from both tables to reduce memory consumption.
- **Use Lookup Tables:** For static or reference data, use lookup tables to reduce computational overhead.

Practical Examples

Let's explore some real-world scenarios where KQL joins can be invaluable.

Example 1: Correlating User Activity

Suppose you have two tables: `UserLogins` and `PageViews`. You want to find users who logged in and visited a specific page.

```
UserLogins
```

```
| where LoginDate >= ago(30d)  
  
| join kind=inner (PageViews | where Page =  
"ProductPage") on UserId
```

This query identifies active users who interacted with a specific page over the last 30 days.

Example 2: Identifying Missing Data

Imagine you have `Orders` and `Shipments` tables, and you need to find orders that have not been shipped.

```
Orders
```

```
| join kind=anti (Shipments) on OrderId
```

This query helps identify gaps in the shipment process that need attention.

Example 3: Combining Error Logs

To analyze error logs from two systems, you can use a full outer join to merge their data.

```
System1Errors
```

```
| join kind=fullouter (System2Errors) on  
ErrorCode
```

This query provides a comprehensive view of errors across both systems.

The join operator in KQL is a versatile and powerful tool for data analysis. By understanding its various types and learning how to optimize and apply them effectively, you can unlock deeper insights and correlations in your data. Whether you're correlating user activity, identifying missing data, or merging error logs, KQL joins can help you achieve your goals with precision and efficiency.

As with any advanced tool, practice is key. Experiment with different join types and scenarios to gain a deeper understanding of how they work and how they can enhance your data queries. Armed with this knowledge, you'll be well-equipped to tackle complex analysis tasks in Azure Monitor and beyond.

- **The Definitive Guide to KQL:** Using Kusto Query Language for operations, defending, and threat hunting <https://amzn.to/42JRsCL>

Chapter 8

Exploring the mv-apply Operator in Kusto Query Language

Kusto Query Language (KQL) is a powerful tool for querying large datasets, and one of its versatile operators is the **mv-apply operator**. This operator allows you to apply a subquery to each record and return the union of the results. Let's dive into its syntax, parameters, and some practical examples to understand its functionality better.

Syntax and Parameters

The **mv-apply operator** can be thought of as a generalization of the **mv-expand operator**. It expands each record in the input into subtables, applies the subquery for each subtable, and returns the union of the results. Here's the basic syntax:

```
T | mv-apply [ItemIndex] ColumnsToExpand  
[RowLimit] on (SubQuery)
```

- **ItemIndex:** Indicates the name of a column that specifies the 0-based index of the element in the array.
- **ColumnsToExpand:** A comma-separated list of expressions that evaluate into dynamic arrays to expand.
- **RowLimit:** Limits the number of records to generate from each input record.
- **SubQuery:** A tabular query expression applied to each array-expanded subtable.

Examples

1. Getting the Largest Element from an Array

This example locates the largest element in each array:

```
let _data = range x from 1 to 8 step 1 |  
summarize l=make_list(x) by xMod2 = x % 2;
```

```
_data | mv-apply element=l to typeof(long) on  
(top 1 by element)
```

Output:

xMod2	I	element
1	[1, 3, 5, 7]	7
0	[2, 4, 6, 8]	8

2. Calculating the Sum of the Largest Two Elements in an Array

This example calculates the sum of the top two elements in each array:

```
let _data = range x from 1 to 8 step 1 |
summarize l=make_list(x) by xMod2 = x % 2;

_data | mv-apply l to typeof(long) on (top 2 by
l | summarize SumOfTop2=sum(l))
```

Output:

xMod2	I	SumOfTop2
1	[1, 3, 5, 7]	12
0	[2, 4, 6, 8]	14

3. Using `with_itemindex` for Working with a Subset of the Array

This example filters elements based on their index:

```
let _data = range x from 1 to 10 step 1 |  
summarize l=make_list(x) by xMod2 = x % 2;
```

```
_data | mv-apply with_itemindex=index element=l  
to typeof(long) on (index >= 3) | project  
index, element
```

Output:

index	element
3	7
4	9
3	8
4	10

The **mv-apply operator** in KQL is a powerful tool for manipulating and querying dynamic arrays.

By understanding its syntax and parameters, and through practical examples, you can leverage this operator to perform complex data transformations efficiently.

Chapter 9

Parsing & Extracting Data from JSON Columns in KQL: Handling Nested Structures Efficiently

JSON (JavaScript Object Notation) has become a universal format for storing and exchanging structured data. It's common to encounter JSON data in nested columns when working with telemetry, logs, or databases. Kusto Query Language (KQL), with its intuitive syntax, provides powerful tools to parse and extract data from JSON columns effectively. This chapter will walk you through the process of working with JSON data in KQL, focusing on nested structures, and provide practical examples.

Why JSON in KQL?

JSON's flexible structure makes it ideal for capturing complex data relationships. In KQL, JSON is often used in columns to store telemetry data from systems like Azure Monitor Logs or Application Insights. Parsing this data efficiently allows analysts to extract valuable insights.

However, handling nested structures can be tricky without a clear roadmap.

Getting Started with JSON in KQL

Before diving into nested structures, it's important to understand how JSON is typically stored and queried in KQL. JSON data is often represented as a text field within a column, and KQL provides functions to parse and work with this data.

Basic JSON Parsing

Let's start with an example JSON column named *EventDetails*:

```
{
```

```
"EventId": 12345,
```

```
"EventType": "Error",
```

```
"Details": {  
    "Source": "System",  
    "Message": "An unexpected error occurred."  
}  
}  
}
```

To extract values from a specific key in this JSON, KQL offers the *parse_json()* function. Here's how you would extract the *EventType*:

```
MyTable
```

```
| extend ParsedEvent = parse_json(EventDetails)  
  
| project EventType = ParsedEvent.EventType
```

In this example:

- *parse_json(EventDetails)* converts the JSON string into an object.
- *ParsedEvent.EventType* accesses the `EventType` field.

Handling Nested Structures

Nested JSON structures can introduce additional layers of complexity. Here's an example of a more complex JSON:

```
{  
  "EventId": 67890,  
  "EventType": "Warning",  
  "Details": {  
    "Source": "Application",  
    "Severity": "Info",  
    "Timestamp": "2023-10-01T12:00:00Z"  
  }  
}
```

```
"Metadata": {  
    "UserId": "U12345",  
    "SessionId": "S67890"  
}  
}  
}
```

To extract deeply nested fields, you can use the dot notation.

Example: Extracting Nested Fields

Suppose you want to extract the *UserId* from the `Metadata` object. The KQL query would look like this:

```
MyTable
```

```
| extend ParsedEvent = parse_json(EventDetails)
```

```
| project UserId =  
ParsedEvent.Details.Metadata.UserId
```

This approach makes it simple to drill down into the nested structure without complex transformations.

Optimizing Performance for Large Datasets

Parsing JSON for every row of a large dataset can impact performance. To handle this efficiently:

- **Use *project* early:** Reduce the number of columns as soon as possible, retaining only the fields you need.
- **Filter first:** Apply *where* filters early in the query to reduce the dataset size before parsing.

- **Leverage *extend* for partial parsing:** If you only need parts of the JSON, limit the parsing scope.

Example: Filtering and Parsing

Here's a query that filters for *Error* events before extracting nested fields:

```
MyTable
```

```
| where EventDetails contains "Error"  
  
| extend ParsedEvent = parse_json(EventDetails)  
  
| project EventId = ParsedEvent.EventId,  
Message = ParsedEvent.Details.Message
```

This approach minimizes the workload by processing only relevant rows.

Flattening Arrays in JSON

JSON often contains arrays, which require flattening for analysis. Consider the following structure:

```
{  
  "LogId": 101,  
  "Timestamp": "2025-05-20T10:57:43Z",  
  "Events": [  
    {"Type": "Info", "Message": "Startup complete"},  
    {"Type": "Error", "Message": "Failed to load configuration"}]  
}
```

```
}
```

To extract data from the *Events* array, use the *mv-expand* operator:

```
MyTable
```

```
| extend ParsedLog = parse_json(LogDetails)
```

```
| mv-expand Events = ParsedLog.Events
```

```
| project LogId = ParsedLog.LogId, EventType = Events.Type, EventMessage = Events.Message
```

Here:

- *mv-expand* creates a row for each element in the array.
- *Events.Type* and *Events.Message* access fields within each array element.

Combining JSON Parsing with Aggregations

Once the data is parsed, you can apply aggregations to summarize or analyze it. For instance, to count the number of *Error* events:

```
MyTable
```

```
| extend ParsedEvent = parse_json(EventDetails)  
  
| where ParsedEvent.EventType == "Error"  
  
| summarize ErrorCount = count()
```

This query efficiently combines parsing with filtering and aggregation for actionable insights.

Best Practices for JSON Parsing in KQL

Here are some additional tips to handle JSON data effectively:

- **Validate JSON formats:** Ensure the data is well-formed to prevent parsing errors.

- **Use aliases:** Rename parsed fields to improve readability and maintain consistency in queries.
- **Document structure:** Maintain a reference of JSON schemas to streamline query writing and debugging.

KQL's built-in functions for JSON parsing and manipulation make it a powerful tool for analyzing nested data structures. By understanding and leveraging techniques like dot notation, filtering, array expansion, and aggregation, you can extract meaningful insights from complex JSON datasets. These skills are especially valuable in scenarios like log analysis, telemetry, and debugging, where JSON is a common format.

With the examples provided, you should feel confident tackling even the most nested JSON structures in KQL. Remember, efficiency and clarity are key—optimize your queries and enjoy the simplicity that KQL brings to your data workflows!

Chapter 10

Parsing & Extracting Data from JSON Columns using KQL: Handling Nested Structures Efficiently

In modern data analytics, JSON (JavaScript Object Notation) is a ubiquitous format for storing and exchanging data due to its flexibility and support for complex, nested structures. When working with JSON data in Azure Data Explorer (ADX) or other platforms that support Kusto Query Language (KQL), efficiently parsing and extracting data from JSON columns is critical for gaining insights. This chapter explores how to handle JSON columns, including nested structures, using KQL. We'll cover key techniques, best practices, and provide working examples to help you master JSON parsing in KQL.

Why JSON in KQL?

JSON is often used to store semi-structured data, such as logs, IoT telemetry, or API responses, in databases. In ADX, JSON data is typically stored

in a column of type dynamic, which allows KQL to parse and query the data flexibly. However, JSON's nested nature can make querying challenging, especially when dealing with arrays, objects, or deeply nested fields. KQL provides powerful operators like **parse_json**, **mv-expand**, and **bag_unpack** to simplify this process.

Key KQL Functions for JSON Parsing

Before diving into examples, let's review the primary KQL functions for working with JSON:

- **parse_json**: Converts a JSON string into a dynamic object for querying.
- **mv-expand**: Expands arrays or property bags into individual rows, useful for nested arrays.
- **bag_unpack**: Flattens a dynamic object (property bag) into separate columns.
- **Dynamic property access**: Use dot notation (e.g., column.field) or bracket notation (e.g., column["field"]) to access JSON properties.
- **todynamic**: An alias for parse_json, used to ensure a string is treated as JSON.

These functions, combined with KQL's query capabilities, make it possible to efficiently extract data from complex JSON structures.

Setting Up a Sample Dataset

To demonstrate JSON parsing, let's use a sample table called DeviceLogs with a JSON column Telemetry. The Telemetry column contains nested JSON data representing device telemetry, including device information, sensor readings, and timestamps.

Here's a sample dataset:

```
let DeviceLogs = datatable(DeviceId: string,  
Telemetry: dynamic)  
  
[  
  
    "DEV001", parse_json('{"device_info":  
        {"name": "SensorA", "location": "Room1"},  
        "readings": [{"timestamp": "2025-05-  
            30T10:00:00Z", "temperature": 22.5, "humidity":  
            45}, {"timestamp": "2025-05-30T10:01:00Z",  
            "temperature": 23.0, "humidity": 47}],  
        "status": "active"}')  
]
```

```
"DEV002", parse_json('{"device_info":  
  {"name": "SensorB", "location": "Room2"},  
  "readings": [{"timestamp": "2025-05-  
    30T10:00:00Z", "temperature": 21.0, "humidity":  
    50}], "status": "inactive"}')  
];
```

This dataset includes:

- A **DeviceId** column (string).
- A Telemetry column (dynamic) with nested JSON containing:
 - **device_info**: An object with name and location.
 - **readings**: An array of objects, each with timestamp, temperature, and humidity.
 - **status**: A string indicating device status.

Let's explore how to parse and extract data from this JSON column.

Example 1: Extracting Top-Level Fields

To extract top-level fields like status from the Telemetry column, you can use dot notation or bracket notation.

Query:

```
DeviceLogs
```

```
| project DeviceId, Status = Telemetry.status
```

Output:

DEVA005	inactive
DEVA001	active
DeviceId	Status

Explanation:

- **Telemetry.status** directly accesses the status field in the JSON object.
- This works for any top-level field in the JSON structure.

Example 2: Accessing Nested Object Fields

To extract fields from a nested object, such as **device_info.name**, chain the property names using dot notation.

Query:

DeviceLogs

```
| project DeviceId, DeviceName =  
Telemetry.device_info.name, Location =  
Telemetry.device_info.location
```

Output:

DEΛ003	sensory	Booms
DEΛ001	Atmos	Boomi
DeviceId	DeviceName	Location

Explanation:

- **Telemetry.device_info.name** navigates the nested **device_info** object to retrieve the name field.
- Multiple fields can be extracted in a single project statement.

Example 3: Handling Nested Arrays with mv-expand

The readings field is an array of objects, which requires special handling. The mv-expand operator is ideal for expanding arrays into rows.

Query:

DeviceLogs

```
| mv-expand Readings = Telemetry.readings
```

```
| project DeviceId, Timestamp =  
Readings.timestamp, Temperature =  
Readings.temperature, Humidity =  
Readings.humidity
```

Output:

DeviceId	Timestamp	Temperature	Humidity
DEV001	2025-05-30T10:00:00Z	22.5	45
DEV001	2025-05-30T10:01:00Z	23.0	47
DEV002	2025-05-30T10:00:00Z	21.0	50

Explanation:

- **mv-expand**
Readings = Telemetry.readings expands the readings array into separate rows, with each row containing one object from the array.
- The Readings alias allows you to reference the expanded objects' fields (e.g., **Readings.timestamp**).

- This is particularly useful for analyzing time-series data or logs stored in arrays.

Example 4: Combining Nested Objects and Arrays

To combine information from nested objects and arrays, use mv-expand and access other fields in the same query.

Query:

```
DeviceLogs
```

```
| mv-expand Readings = Telemetry.readings  
  
| project DeviceId, DeviceName =  
Telemetry.device_info.name, Timestamp =  
Readings.timestamp, Temperature =  
Readings.temperature, Status = Telemetry.status
```

Output:

DeviceId	DeviceName	Timestamp	Temperature	Status
DEV001	SensorA	2025-05-30T10:00:00Z	22.5	active
DEV001	SensorA	2025-05-30T10:01:00Z	23.0	active
DEV002	SensorB	2025-05-30T10:00:00Z	21.0	inactive

Explanation:

- **mv-expand** handles the readings array, while **Telemetry.device_info.name** and **Telemetry.status** access fields from the original JSON.
- The result includes both array-derived fields (e.g., Temperature) and top-level or nested fields (e.g., DeviceName, Status).

Example 5: Flattening JSON with **bag_unpack**

For JSON objects with many fields, **bag_unpack** can flatten a dynamic object into columns automatically.

Query:

DeviceLogs

```
| project DeviceId, DeviceInfo =
Telemetry.device_info
```

```
| evaluate bag_unpack(DeviceInfo)
```

Output:

DeviceId	name	location
DEV001	SensorA	Room1
DEV002	SensorB	Room2

Explanation:

- **project DeviceId, DeviceInfo = Telemetry.device_info** extracts the **device_info** object.
- **evaluate bag_unpack(DeviceInfo)** converts the **device_info** object's fields (name, location) into separate columns.
- This is useful when you don't know the JSON structure in advance or want to avoid manually specifying fields.

Example 6: Handling Missing or Null Fields

JSON data often has missing or null fields, which can cause errors if not handled. Use the coalesce function or conditional checks to manage this.

Query:

DeviceLogs

```
| project DeviceId, ErrorCode =  
coalesce(Telemetry.error_code, "NoError")
```

Output:

DeviceId	ErrorCode
DEV001	NoError
DEV002	NoError

Explanation:

- Since **error_code** doesn't exist in the JSON, coalesce returns the fallback value "**NoError**".
- This ensures queries remain robust against missing fields.

Best Practices for Efficient JSON Parsing in KQL

1. **Use parse_json Sparingly:** If the JSON is already stored as a dynamic column, you don't need parse_json. Use it only for string columns containing JSON.

2. **Leverage mv-expand for Arrays:** Always use mv-expand to handle arrays, as it simplifies querying nested data.
3. **Optimize with project:** Reduce the dataset early with project to include only necessary columns, improving performance.
4. **Handle Missing Data:** Use coalesce, isnull, or isnotnull to manage missing or null fields gracefully.
5. **Use bag_unpack for Dynamic Structures:** When dealing with unknown or variable JSON schemas, bag_unpack can save time.
6. **Test with Small Datasets:** Before running queries on large datasets, test with a small sample to ensure correctness.
7. **Index JSON Columns (if possible):** In ADX, consider extracting frequently queried JSON fields into materialized views or computed columns for faster access.

Performance Considerations

Parsing JSON in KQL is generally efficient, but large datasets or deeply nested structures can impact performance. Here are tips to optimize:

- **Filter Early:** Apply where clauses before parsing JSON to reduce the dataset size.
- **Avoid Over-Expanding:** Be cautious with mv-expand on large arrays, as it can generate many rows. Use take or limit to test.
- **Cache Results:** For frequently accessed JSON data, consider pre-processing it into a table with extracted fields using ADX's materialized views.
- **Monitor Query Cost:** Use the .show queries command in ADX to analyze query performance and optimize as needed.

KQL provides a robust set of tools for parsing and extracting data from JSON columns, even when dealing with complex nested structures. By mastering functions like parse_json, mv-expand, and bag_unpack, you can efficiently query JSON data to uncover valuable insights. The examples in this post demonstrate how to handle top-level fields, nested objects, arrays, and dynamic structures, while best practices ensure your queries are both effective and performant.

Whether you're analyzing IoT telemetry, application logs, or API responses, KQL's JSON

parsing capabilities make it a powerful tool for data exploration. Try these techniques in your own ADX environment, and experiment with your JSON data to see how KQL can streamline your workflows.

Chapter 11

Harnessing KQL for Machine Learning Data Prep in Azure Data Explorer

Data preparation is the backbone of any successful machine learning (ML) project, often consuming the lion's share of time and effort. In Azure, combining **Kusto Query Language (KQL)** with **Azure Data Explorer (ADX)** provides a powerful, scalable solution for cleaning and transforming data before feeding it into **Azure Machine Learning (AML)**. In this post, we'll explore how KQL can streamline data prep tasks and seamlessly integrate with AML for robust ML workflows.

Why KQL and Azure Data Explorer?

Azure Data Explorer is a fast, fully managed data analytics service optimized for real-time analysis of large datasets. Its query language, KQL, is intuitive yet powerful, making it ideal for data wrangling tasks like filtering, aggregating, and feature engineering. By preparing data in ADX, you can leverage its scalability and then export clean datasets to AML for model training.

Scenario: Preparing IoT Sensor Data for Predictive Maintenance

Let's consider a common ML use case: predicting equipment failures using IoT sensor data. Our dataset, stored in ADX, contains sensor readings with columns for DeviceId, Timestamp, Temperature, Pressure, and OperationalStatus. We need to clean and transform this data before training a model in AML.

Step 1: Setting Up the Data

Assume our table in ADX is called SensorData. A sample record looks like this:

DeviceId	Timestamp	Temperature	Pressure	OperationalStatus
D001	2025-06-01T10:00:00Z	75.2	101.5	Normal
D001	2025-06-01T10:01:00Z	80.1	null	Normal
D002	2025-06-01T10:00:00Z	null	99.8	Failed

Our goal is to:

- Handle missing values.
- Filter out irrelevant data.
- Engineer features (e.g., rolling averages).
- Export the cleaned dataset to AML.

Step 2: Cleaning Data with KQL

Let's write KQL queries to clean the data.

Handling Missing Values

Missing Temperature or Pressure values can skew ML models. We can impute missing values with the column's median or remove rows with nulls. Here's a KQL query to impute missing Pressure with the median:

```
let MedianPressure = toscalar(  
    SensorData  
        | summarize MedianPressure =  
            percentile(Pressure, 50)  
        | where isnotnull(MedianPressure)  
) ;
```

```
SensorData
```

```
| extend Pressure = coalesce(Pressure,  
MedianPressure)
```

For Temperature, let's remove rows where it's missing, as it's critical for our model:

```
SensorData
```

```
| where isnotnull(Temperature)
```

Filtering Irrelevant Data

Suppose we only want data from the last 30 days and devices with OperationalStatus as "Normal" or "Failed" (excluding maintenance states). Here's the query:

```
SensorData
```

```
| where Timestamp > ago(30d)
```

```
| where OperationalStatus in ("Normal",  
"Failed")
```

Step 3: Feature Engineering with KQL

ML models often benefit from derived features.
Let's create a rolling average of Temperature over
a 5-minute window for each device to capture
trends:

```
SensorData
```

```
| where isnotnull(Temperature)
```

```
| order by DeviceId, Timestamp
```

```
| extend RollingAvgTemp =  
series_fir(Temperature, array_repeat(1, 5),  
true, 5)
```

We can also encode OperationalStatus as a binary
label (0 for Normal, 1 for Failed) for classification:

```
SensorData
```

```
| extend Label = iff(OperationalStatus ==  
"Failed", 1, 0)
```

Step 4: Aggregating and Summarizing

To reduce noise, let's aggregate data into hourly summaries, calculating average Temperature and Pressure per DeviceId:

```
SensorData
```

```
| where Timestamp > ago(30d)
```

```
| where isnotnull(Temperature)
```

```
| summarize AvgTemp = avg(Temperature),  
AvgPressure = avg(Pressure), MaxLabel =  
max(Label)
```

```
by DeviceId, bin(Timestamp, 1h)
```

This creates a cleaner dataset with hourly aggregates, ready for ML.

Step 5: Exporting to Azure Machine Learning

Once the data is prepped, we need to export it from ADX to a format AML can use (e.g., CSV or Parquet). ADX supports exporting query results to Azure Blob Storage:

```
.export  
to csv (  
    h@"https://<storage-  
account>.blob.core.windows.net/<container>/clea  
ned_data.csv;<storage-key>"  
)  
with (
```

```
includeHeaders="all",  
  
encoding="utf8"  
  
)  
  
  
  
select DeviceId, Timestamp, AvgTemp,  
AvgPressure, MaxLabel  
  
  
  
from SensorData  
  
  
  
  
| where Timestamp > ago(30d)  
  
  
  
| where isnotnull(Temperature)  
  
  
  
| summarize AvgTemp = avg(Temperature),  
AvgPressure = avg(Pressure), MaxLabel =  
max(Label)
```

```
by DeviceId, bin(Timestamp, 1h)
```

In AML, you can then:

1. Create a **Datastore** pointing to your Blob Storage.
2. Register the exported file as a **Dataset**.
3. Use the dataset in an AML pipeline for model training.

Alternatively, use the **Azure Data Explorer connector** in AML to query ADX directly, though exporting to Blob is often simpler for static datasets.

Step 6: Automating the Pipeline

To productionize, automate the data prep process:

- Schedule KQL queries using **Azure Data Factory** or **Logic Apps** to run periodically.
- Trigger an AML pipeline to retrain the model whenever new data is exported.
- Monitor data quality with ADX dashboards to ensure consistency.

Benefits of Using KQL for ML Data Prep

- **Scalability:** ADX handles massive datasets with ease, outperforming

traditional SQL databases for time-series data.

- **Simplicity:** KQL's syntax is concise, reducing the learning curve for data engineers.
- **Integration:** Seamless connectivity with AML via Blob Storage or direct connectors.
- **Flexibility:** KQL supports advanced transformations like series analysis, ideal for IoT and time-series use cases.

KQL in Azure Data Explorer is a game-changer for preparing data for machine learning. Its ability to clean, transform, and aggregate large datasets efficiently makes it a natural fit for AML workflows. By following the steps outlined—cleaning data, engineering features, and exporting to AML—you can build robust, scalable ML pipelines with ease.

Chapter 12

Let's Get Scalar: Creating Reusable KQL Functions for Cleaner Queries

When working with large datasets in Azure Data Explorer, Log Analytics, or other platforms that support Kusto Query Language (KQL), queries can quickly become complex and unwieldy. Repeated logic, nested subqueries, and sprawling code make maintenance a nightmare. Enter **KQL functions**—a powerful way to encapsulate logic, improve readability, and make your queries reusable. In this post, we'll explore how to create reusable KQL functions, why they matter, and provide practical examples to help you write cleaner, more maintainable queries.

Why Use KQL Functions?

KQL functions allow you to:

- **Encapsulate logic:** Package repetitive or complex logic into a single, reusable unit.
- **Improve readability:** Break down large queries into modular components.

- **Enhance maintainability:** Update logic in one place instead of multiple queries.
- **Promote consistency:** Ensure the same logic is applied uniformly across queries.

Think of functions as your personal query-building blocks. Instead of rewriting the same filtering or aggregation logic, you define it once and call it whenever needed.

Types of KQL Functions

KQL supports two main types of functions:

1. **User-Defined Functions (UDFs):** Custom functions you create to encapsulate logic, stored in the database.
2. **Let Statements:** Inline functions defined within a query for one-time or query-specific use.

We'll cover both, with a focus on UDFs for reusability across queries.

Creating a User-Defined Function (UDF)

To create a UDF, you use the `.create function` command in KQL. The syntax is:

```
.create function FunctionName([parameters]) {  
    // Logic here  
}
```

Example 1: Filtering High-Severity Alerts

Let's say you frequently query a SecurityAlerts table to find alerts with a severity of "High" or "Critical" from the last 7 days. Without a function, you'd repeat this logic in every query:

```
SecurityAlerts  
  
| where TimeGenerated > ago(7d)  
  
| where Severity in ("High", "Critical")
```

To make this reusable, create a UDF:

```
.create function HighSeverityAlerts() {  
    SecurityAlerts  
        | where TimeGenerated > ago(7d)  
        | where Severity in ("High", "Critical")  
}  
}
```

Now, you can call this function in any query:

```
HighSeverityAlerts()  
  
| summarize count() by AlertName
```

This function is stored in the database and can be reused across queries, reducing duplication and making updates easier. If you need to change the time range to 14 days, you update the function once, and all queries using it reflect the change.

Example 2: Parameterized Function for Flexible Filtering

Functions become even more powerful with parameters. Let's create a function that filters alerts by a custom time range and severity level.

```
.create function
FilterAlerts(TimeRange:timespan,
SeverityLevel:string) {

    SecurityAlerts

        | where TimeGenerated > ago(TimeRange)

        | where Severity == SeverityLevel

}
```

Use it like this:

```
FilterAlerts(7d, "High")
```

```
| summarize count() by AlertName
```

Or with a different time range and severity:

```
FilterAlerts(1h, "Critical")
```

```
| project AlertName, TimeGenerated
```

Parameters make functions flexible, allowing you to reuse the same logic with different inputs.

Using Let Statements for Inline Functions

For one-off or query-specific logic, you can use let statements to define inline functions. These aren't stored in the database but are great for breaking down complex queries within a single session.

Example 3: Inline Function for Aggregation

Suppose you're analyzing web server logs and want to calculate average response times by endpoint. You might need to clean and

preprocess the data repeatedly. Use a let statement:

```
let CleanWebLogs = () {  
    WebLogs  
    | where ResponseTime > 0  
    | extend Endpoint = tolower(Endpoint)  
};  
  
CleanWebLogs()  
  
| summarize AvgResponseTime = avg(ResponseTime)  
by Endpoint
```

You can also add parameters:

```
let CleanWebLogs = (MinResponseTime:real) {  
    WebLogs  
    | where ResponseTime > MinResponseTime  
    | extend Endpoint = tolower(Endpoint)  
};  
  
CleanWebLogs(100)
```

| summarize AvgResponseTime = avg(ResponseTime)
by Endpoint

let statements are scoped to the query, so they're perfect for modularizing logic without cluttering the database with temporary functions.

Best Practices for KQL Functions

1. **Use Descriptive Names:** Name functions clearly (e.g., HighSeverityAlerts instead of Func1) to convey their purpose.
2. **Keep Functions Focused:** Each function should have a single, well-defined responsibility.
3. **Leverage Parameters:** Make functions flexible with parameters for time ranges, thresholds, or other variables.
4. **Document Functions:** Use comments or metadata to describe what the function does and its parameters.
5. **Test Before Saving:** Validate UDF logic in a query before creating the function to avoid errors.
6. **Manage Permissions:** Ensure appropriate access controls for UDFs in shared environments.

Real-World Example: Combining Functions for Complex Queries

Let's combine what we've learned to tackle a more complex scenario. You're monitoring a system with multiple data sources: SecurityAlerts and SystemLogs. You want to identify high-severity alerts and correlate them with system errors from the same time period.

First, create a UDF for high-severity alerts (as shown earlier):

```
.create function HighSeverityAlerts() {  
    SecurityLogs  
    | where TimeGenerated > ago(7d)  
    | where Severity in ("High", "Critical")  
}  
}
```

Next, create a UDF for system errors:

```
.create function  
SystemErrors (TimeRange:timespan) {  
    SystemLogs
```

```
| where TimeGenerated > ago(TimeRange)  
  
| where LogLevel == "Error"  
  
}
```

Now, combine them in a query:

```
HighSeverityAlerts()
```

```
| join kind=inner (  
  
    SystemErrors(7d)  
  
) on $left.TimeGenerated ==  
$right.TimeGenerated  
  
| project AlertName, LogMessage, TimeGenerated
```

This query is clean, readable, and maintainable because the logic is encapsulated in reusable functions. If you need to adjust the severity levels or error conditions, you update the respective functions without touching the main query.

KQL functions—whether user-defined or inline let statements—are a game-changer for managing complex queries. By encapsulating logic, you reduce duplication, improve readability, and make your queries easier to maintain. Start small with let statements for query-specific needs, then graduate to UDFs for reusable, database-stored functions. With parameters and best practices, you can build a library of functions that streamline your KQL workflows.

Try creating a few functions for your most common query patterns. You'll be amazed at how much cleaner and more efficient your KQL queries become!

Chapter 13

Mastering Row-Level Security and Access Controls in KQL: Enterprise Strategies for Multi-Tenant Isolation and Auditing

In the world of big data analytics, Azure Data Explorer (ADX) and its Kusto Query Language (KQL) shine as a powerhouse for handling massive datasets in real-time. But for enterprises running shared clusters—think multi-tenant SaaS platforms or consolidated analytics environments—the real challenge isn't just querying data; it's securing it at the granular level. Row-level security (RLS) and query-time access controls ensure that tenants see only their own data, while audit trails keep compliance teams happy. These features are enterprise-grade, yet they're often buried in docs or overlooked in tutorials. Today, we'll dive deep into implementing function-based row filters for ironclad multi-tenant isolation and query patterns for auditing access in shared setups. Buckle up—this is KQL at its most secure.

Why Row-Level Security Matters in Enterprise KQL

Enterprises love ADX for its scalability, but shared clusters amplify risks: a misconfigured query could leak sensitive tenant data. RLS addresses this by enforcing filters at the database layer, transparent to users and apps. Unlike application-layer checks, RLS is tamper-proof—even admins are bound by it. For multi-tenant scenarios, it enables logical data silos in a single table, slashing costs while boosting isolation. And for auditing? KQL's extensibility lets you log and query access patterns, feeding into SIEM tools or compliance dashboards.

Let's break it down with practical implementations.

Implementing Function-Based Row Filters for Multi-Tenant Data Isolation

The cornerstone of RLS in KQL is the `.alter table` policy `row_level_security` command. It applies a filtering query (or function) to a table, restricting rows based on the executing principal's identity—perfect for tenant isolation. The magic happens

with reusable functions: define once, apply everywhere, and leverage `current_principal()` or Entra ID groups to match tenant IDs dynamically.

Step 1: Design Your Tenant Filter Function

Start by creating a function that inspects the user's context and filters rows. Assume a `TenantId` column in your tables (e.g., `Orders` or `Logs`). Use `current_principal_details()` to extract the user's email or group, map it to a tenant, and filter.

Here's a reusable function for multi-tenant filtering:

```
.create-or-alter function
MultiTenantFilter(TableName: string) {

    let UserTenant =
        // Map user email to tenant (replace
        // with your lookup logic, e.g., from a config
        // table)
}
```

```
case (

current_principal_details()["UserPrincipalName"]
] has "tenantA.com", "A",

current_principal_details()["UserPrincipalName"]
] has "tenantB.com", "B",

"Default" // Fallback for admins

);

table(TableName)

| where TenantId == UserTenant or
current_principal_is_member_of('aadgroup=global-
admins@yourorg.com')
```

```
// Optional: Mask sensitive columns for
non-admins

    | extend SensitiveCol =
iff(current_principal_is_member_of('aadgroup=global-admins@yourorg.com'), SensitiveCol,
"REDACTED")

}
```

This function:

- Derives the tenant from the user's UPN (User Principal Name).
- Filters rows to match the tenant.
- Grants full access (no masking) to global admins via Entra group check.
- Handles edge cases with assert() if needed: | where assert(UserTenant != "", "Unauthorized tenant access").

For more sophisticated mapping, join against a TenantMappings table:

```
let TenantMappings = TenantMappingsTable
```

```
| where UserEmail ==  
current_principal_details()["UserPrincipalName"  
]  
  
| project TenantId;  
  
  
  
  
table(TableName)  
  
  
  
  
| where TenantId in (TenantMappings)
```

Step 2: Apply the Policy to Tables

Enable RLS on your tables with a one-liner per table. This replaces default access—queries now run through your filter automatically.

```
.alter table Orders policy row_level_security  
enable "MultiTenantFilter('Orders')"
```

```
.alter table CustomerLogs policy
row_level_security enable
"MultiTenantFilter('CustomerLogs')"
```

Test it: A user from tenantA.com querying Orders sees only TenantId “A” rows. Cross-tenant peeking? Blocked at the engine level.

Pro Tips for Multi-Tenant Scale

- **Performance:** Index on TenantId to keep filters blazing fast.
- **Group-Based Access:** For RBAC-heavy setups, chain current_principal_is_member_of() for role-tiered views (e.g., tenant admins see all, reps see subsets).
- **Limitations:** RLS can't reference other RLS-enabled tables or cross-databases—keep lookups lightweight.
- **Rollback:** .delete table Orders policy row_level_security if things go sideways.

This setup has saved enterprises millions in separate cluster costs while maintaining HIPAA/GDPR isolation.

Query Patterns for Auditing Access in Shared Clusters

In shared clusters, “who saw what?” isn’t optional—it’s audit fodder. KQL doesn’t have built-in query auditing, but Azure Monitor’s LAQueryLogs table captures it all: who ran a query, what text, performance, and targets. Pair this with the restrict statement for query-time controls, and you’ve got proactive access enforcement plus reactive logging.

Pattern 1: Enforce Access with the restrict Statement

The restrict access to statement is your middle-tier enforcer—ideal for apps proxying user queries. It hides unauthorized entities (tables/views) from the query scope, implementing RLS without policies.

Example for a shared analytics dashboard:

```
// Middle-tier prepends this based on user context
```

```
let TenantView = view () {  
  
    AllData | where TenantId ==  
    extract_tenant_from_user() // Your tenant  
    logic  
  
};  
  
restrict access to (TenantView);  
  
// User's query now scoped  
  
TenantView  
  
| summarize Count = count() by bin(Timestamp,  
1h)
```

```
| render timechart
```

This confines the user to TenantView—attempts to query AllData directly fail invisibly. For shared clusters, restrict cross-database: restrict access to (database('SharedDB').TenantView);.

Audit hook: Log the restricted query text to a custom table for correlation.

Pattern 2: Auditing Queries via LAQueryLogs

Enable query auditing in your Log Analytics workspace (Azure Portal > Diagnostic settings > Send to Log Analytics). Then, KQL-analyze access patterns:

Basic who/when/what:

```
LAQueryLogs
```

```
| where TimeGenerated > ago(7d)
```

```
| where RequestClientApp == "AzureDataExplorer"  
// Filter for ADX queries
```

```
| project TimeGenerated, AADEmail, QueryText,  
ResponseDurationMs
```

```
| summarize QueryCount = count(), AvgDuration =  
avg(ResponseDurationMs) by AADEmail,  
bin(TimeGenerated, 1d)
```

```
| render barchart
```

Spot anomalies in shared access:

```
LAQueryLogs
```

```
| where TimeGenerated > ago(30d)
```

```
| where AADTenantId == "your-tenant-id" //  
Shared cluster filter
```

```
| extend TablesAccessed =  
extract(@"table\s*\\"(.*)\"\")?", 1, QueryText)  
// Parse tables from query text
```

```
| where TablesAccessed has "SensitiveTable" //  
Flag high-risk access  
  
| summarize AccessEvents = count() by AADEmail,  
TablesAccessed  
  
| where AccessEvents > 10 // Threshold for  
alerts  
  
| render table
```

This reveals over-access (e.g., a user hammering a tenant's table) or unusual patterns (queries at 3 AM). Integrate with Sentinel for alerts: | where AccessEvents > threshold | invoke AlertRule.

Advanced Auditing Twist

Cross-correlate with Entra sign-ins:

```
union LAQueryLogs, SigninLogs
```

```
| where TimeGenerated > ago(1d)
```

```
| where AADEmail == "suspicious@tenant.com"  
  
| project TimeGenerated, ActivityType,  
QueryText or IPAddress  
  
| order by TimeGenerated desc
```

In shared clusters, pipe these to a dedicated audit table via continuous export for long-term retention.

Secure, Scalable KQL for the Enterprise

Row-level security via function-based filters turns multi-tenant chaos into compliant harmony, while restrict and LAQueryLogs patterns keep your shared clusters auditable and airtight. These aren't beginner tricks—they're the glue for petabyte-scale enterprises. Start small: Prototype on a dev cluster, monitor with the audits above, and scale.

For deeper dives, check the [official RLS docs](#) or [restrict statement guide](#). Stay secure out there.

Chapter 14

Similarity and Distance Metrics in KQL Queries: Advanced Techniques for Threat Intelligence Matching

Exact matches aren't always enough. Attackers often use subtle variations in URLs, domains, or indicators of compromise (IOCs) to evade detection—think phishing sites with slightly altered domain names or mutated malware signatures. This is where similarity and distance metrics come into play. Kusto Query Language (KQL), the powerhouse behind Azure Data Explorer, Azure Monitor, and Microsoft Sentinel, offers built-in tools for set-based similarity and can be extended for more advanced string comparisons.

In this post, we'll explore how to leverage KQL's `jaccard_index` function combined with `to_utf8` (or its modern alias `unicode_codepoints_from_string`) for detecting similar URLs or IOCs in scenarios like web crawling or phishing detection. We'll also discuss extending KQL with Levenshtein distance or custom fuzzy matching using Python plugins for

entity resolution tasks, such as matching user names or file hashes across datasets.

Whether you're a threat hunter, data analyst, or security engineer, these techniques can level up your queries for more resilient detections.

Understanding the Jaccard Index in KQL

The Jaccard index, also known as Jaccard similarity coefficient, measures the similarity between two finite sets. It's calculated as the size of the intersection divided by the size of the union of the sets:

Values range from 0 (no similarity) to 1 (identical sets). In KQL, the `jaccard_index` function takes two dynamic arrays as inputs and computes this metric, treating the arrays as sets (duplicates are ignored).

This is particularly useful for threat intelligence because IOCs like URLs or domains can be broken down into sets of characters, n-grams (substrings

of length n), or tokens. For example, in phishing detection, you might compare a suspicious URL against known malicious ones to flag high-similarity variants.

Basic Syntax and Example

Here's the basic syntax:

```
jaccard_index(set1: dynamic, set2: dynamic)
```

A simple example with numeric sets:

```
print set1 = dynamic([1, 2, 3]), set2 =  
dynamic([2, 3, 4])
```

```
| extend similarity = jaccard_index(set1, set2)
```

Output: similarity = 0.5 (intersection: {2,3}; union: {1,2,3,4}).

Using Jaccard Index with to_utf8 for URL or IOC Similarity

To apply Jaccard to strings like URLs, convert them to arrays of Unicode codepoints using `to_utf8` (deprecated alias for `unicode_codepoints_from_string`). This treats the string as a set of characters, allowing you to compute character-level similarity.

Why `to_utf8`? It encodes the string into an array of integers representing Unicode codepoints, which `jaccard_index` can handle directly.

Example: Basic String Similarity

Suppose you're analyzing web logs for phishing attempts. Compare a suspicious URL like "micros0ft-login.com" against a known one "microsoft-login.com":

```
let suspicious_url = "micros0ft-login.com";
```

```
let known_url = "microsoft-login.com";
```

```
print suspicious_chars =
  to_utf8(suspicious_url),
```

```
known_chars = to_utf8(known_url)

| extend similarity =
jaccard_index(suspicious_chars, known_chars)
```

This might yield a high similarity score (e.g., ~0.85) due to shared characters, flagging it for review.

In a real query over a table:

WebLogs

```
| where Url contains "login"

| extend url_chars = to_utf8(Url)

| extend known_malicious_chars =
to_utf8("evilphish.com")

| extend similarity = jaccard_index(url_chars,
known_malicious_chars)
```

```
| where similarity > 0.7  
  
| project Url, similarity
```

This could detect variants in web crawler data or proxy logs.

Enhancing with N-Grams for Better Accuracy

Character-level Jaccard works for rough matches but ignores order. For more robust URL/IOC similarity (e.g., in phishing detection), use *shingling*: break strings into overlapping n-grams and compute Jaccard on those sets.

KQL doesn't have a built-in n-gram function, but you can generate them using range and substring:

```
let url = "microsoft.com";
```

```
let n = 3; // trigram
```

```
let shingles = pack_array(  
      
    substring(url, 0, n),  
      
    substring(url, 1, n),  
      
    substring(url, 2, n),  
      
    // ... extend for full length using mv-  
    apply or a loop  
      
    substring(url, strlen(url) - n, n)  
);
```

For dynamic generation in a query:

```
let get_shingles = (s: string, n: int) {
```

```
range(0, strlen(s) - n, 1)

| mv-apply start = Column1 to typeof(int)
on (
    extend shingle = substring(s, start, n)

    | summarize shingles =
make_set(shingle)

)

| project shingles

};

let suspicious =
toscalar(get_shingles("microsoft.com", 3));
```

```
let known =  
toscalar(get_shingles("microsoft.com", 3));  
  
print similarity = jaccard_index(suspicious,  
known)
```

This captures sequential similarity better, useful for detecting typo-squatted domains in threat intel feeds.

In practice, apply this in Microsoft Sentinel for hunting similar IOCs across SigninLogs or SecurityEvents.

Extending to Levenshtein Distance and Custom Fuzzy Matching via Plugins

While Jaccard excels at set-based similarity, it doesn't account for edit operations (insertions, deletions, substitutions). That's where Levenshtein distance (edit distance) shines: it quantifies the minimum changes needed to transform one string into another.

KQL lacks a native Levenshtein function, but you can implement it using the Python plugin. This

runs sandboxed Python code as a user-defined function (UDF), leveraging libraries like NumPy and Pandas (pre-installed in the sandbox).

Enabling and Using the Python Plugin

First, ensure the Python plugin is enabled in your Azure Data Explorer cluster (it's on by default in many setups). The syntax for a Python UDF:

```
.create function MyLevenshtein(str1: string,  
str2: string): (distance: int)
```

```
{
```

```
    python(
```

```
        nameof(str1), nameof(str2), nameof(distance),
```

```
def levenshtein(s1, s2):
    if len(s1) < len(s2):
        return levenshtein(s2, s1)
    if len(s2) == 0:
        return len(s1)
    previous_row = range(len(s2) + 1)
    for i, c1 in enumerate(s1):
        current_row = [i + 1]
        for j, c2 in enumerate(s2):
            insertions = previous_row[j + 1] + 1
            deletions = current_row[j] + 1
            substitutions = previous_row[j] + (c1 != c2)
            current_row.append(min(insertions, deletions, substitutions))
        previous_row = current_row
    return previous_row[-1]

df['distance'] = df.apply(lambda row: levenshtein(row['str1'], row['str2']), axis=1)
```

)

}

This defines a pure-Python Levenshtein implementation (no external libs needed).

Example: Entity Resolution with Levenshtein

For resolving similar entities (e.g., user names in logs):

```
UserLogs
```

```
| extend known_user = "john.doe@example.com"  
  
| invoke MyLevenshtein(UserEmail, known_user)  
  
| where distance < 3 // Allow up to 2 edits  
(typos)  
  
| project UserEmail, distance
```

This could match "jhon.doe@example.com" or "john.d0e@example.com" in threat hunting for account compromise.

Custom Fuzzy Matching

For more advanced fuzzy logic (e.g., combining Levenshtein with phonetic matching like Soundex), extend the Python script:

- Use fuzzy library? The sandbox doesn't include it, so stick to built-ins or implement.
- Or integrate with NumPy for vectorized distances.

Example enhancement:

Add a fuzzy ratio in the Python code using a normalized Levenshtein ($1 - \text{distance} / \text{max_length}$).

This plugin approach is powerful for entity resolution in large datasets, like merging IOC lists from different sources.

Practical Applications in Threat Intelligence

- **Phishing Detection:** Use Jaccard on URL shingles to flag similar domains in email logs.
- **IOC Matching:** Compare file hashes or IP ranges with fuzzy thresholds to catch variants.

- **Entity Resolution:** Deduplicate threat actor aliases or user identities across intel feeds.

Remember, high similarity doesn't always mean malice—tune thresholds based on your data to avoid false positives.

KQL's jaccard_index with to_utf8 provides a lightweight way to implement set-based similarity for quick threat matching, while Python plugins unlock advanced metrics like Levenshtein for deeper analysis. These tools make KQL a versatile language for cybersecurity workflows.

Experiment in your Azure environment, and share your detections! For more on KQL functions, check the official docs.

Note: Code examples are illustrative; test in your setup for performance on large datasets.

Chapter 15

Statistical Aggregations for Anomaly Detection Using KQL

Anomaly detection is a critical tool for identifying unusual patterns that could signal issues like security breaches, system failures, or operational inefficiencies. While basic aggregations like averages (using `avg()`) are commonplace in introductory KQL tutorials, they often fall short for robust outlier identification. Kusto Query Language (KQL), the querying powerhouse behind Azure Data Explorer and Azure Monitor, offers advanced statistical functions such as `stdev()`, `variance()`, `percentiles_array()`, and `avgif()` to delve deeper into data distributions. These help quantify variability and condition-based summaries, enabling more nuanced anomaly detection.

Beyond static summaries, KQL shines in time-series analysis with functions like `series_decompose_anomalies()`, which decomposes data into trends, seasonal patterns, and residuals to spot deviations in monitoring

pipelines. This post explores these tools, with practical examples applied to metrics like network traffic and login attempts—scenarios where outliers might indicate threats or anomalies.

Beyond Averages: Key Statistical Aggregations in KQL

Simple averages provide a central tendency but ignore spread and conditions. Advanced aggregations address this by measuring dispersion (via standard deviation and variance), distribution points (percentiles), and conditional averages. These are typically used with the summarize operator to group and compute metrics.

Standard Deviation (`stdev()`) and Variance (`variance()`)

Standard deviation measures how much values deviate from the mean, while variance is its squared form—both are essential for identifying outliers. High variance indicates erratic behavior, perfect for flagging unusual spikes in metrics.

- **Syntax for stdev():** `stdev(expression)`
Returns the standard deviation of the expression across the group.
- **Syntax for variance():**
`variance(expression)` Returns the variance of the expression across the group.

Consider monitoring login attempts in `SigninLogs` (a common table in Azure Monitor). To detect anomalies, calculate the average and standard deviation of attempts per hour, then flag hours where attempts exceed the average plus three standard deviations (a common threshold for outliers):

```
SigninLogs
```

```
| where TimeGenerated > ago(1d)
```

```
| summarize AvgAttempts = avg(count()),  
StdevAttempts = stdev(count()),  
VarianceAttempts = variance(count()) by  
bin(TimeGenerated, 1h)
```

```
| extend UpperThreshold = AvgAttempts + 3 *  
StdevAttempts  
  
| where count_ > UpperThreshold  
  
| project TimeGenerated, count_, AvgAttempts,  
StdevAttempts, VarianceAttempts, UpperThreshold
```

This query groups logins by hour, computes the metrics, and identifies potential brute-force attacks if attempts spike unusually. Similarly, for network traffic in CommonSecurityLog:

CommonSecurityLog

```
| where TimeGenerated > ago(1d)  
  
| summarize AvgBytes = avg(SentBytes +  
ReceivedBytes), StdevBytes = stdev(SentBytes +  
ReceivedBytes), VarianceBytes =  
variance(SentBytes + ReceivedBytes) by  
bin(TimeGenerated, 5m)
```

```
| extend AnomalyThreshold = AvgBytes + 3 *  
StdevBytes  
  
| where (SentBytes + ReceivedBytes) >  
AnomalyThreshold  
  
| project TimeGenerated, TotalBytes = SentBytes  
+ ReceivedBytes, AvgBytes, StdevBytes,  
VarianceBytes
```

High variance here might reveal data exfiltration or DDoS attempts.

Percentiles (`percentiles_array()`)

Percentiles help understand data distribution by showing values below which a given percentage of observations fall. `percentiles_array()` returns an array of multiple percentiles, useful for spotting outliers in skewed datasets.

- **Syntax:** `percentiles_array(expression, percentiles)` Where `percentiles` is an array of values (0-100). Returns an array of approximates.

For login attempts, compute the 95th and 99th percentiles to identify extreme values:

SigninLogs

```
| where TimeGenerated > ago(7d)

| summarize Percentiles =
percentiles_array(count(), dynamic([95, 99]))
by UserPrincipalName

| mv-expand Percentile95 = Percentiles[0],
Percentile99 = Percentiles[1]

| where count_ > Percentile99

| project UserPrincipalName, count_,
Percentile95, Percentile99
```

This flags users with login counts in the top 1%, potentially indicating compromised accounts. For network traffic:

NetworkLogs

```
| summarize PercentilesTraffic =  
percentiles_array(DataTransferred, dynamic([50,  
90, 99])) by bin(TimeGenerated, 1h)  
  
| mv-expand Median = PercentilesTraffic[0], P90 =  
PercentilesTraffic[1], P99 =  
PercentilesTraffic[2]  
  
| where DataTransferred > P99  
  
| project TimeGenerated, DataTransferred,  
Median, P90, P99
```

Outliers above the 99th percentile could signal unusual bursts.

Conditional Averaging (avgif())

avgif() computes averages only for rows meeting a condition, ideal for focused analysis like successful vs. failed logins.

- **Syntax:** avgif(expression, predicate)
Averages the expression where the predicate is true.

Example for login attempts, averaging only failed ones:

SigninLogs

```
| summarize FailedAvg = avgif(count(),  
ResultType == "Failure") by bin(TimeGenerated,  
1h)
```

```
| where FailedAvg > 50 // Custom threshold for  
alerting
```

For network traffic, average bytes only for outbound traffic:

CommonSecurityLog

```
| summarize OutboundAvg = avgif(SentBytes,  
CommunicationDirection == "OUTBOUND") by  
DeviceVendor
```

```
| where OutboundAvg > 1000000 // Flag high  
outbound averages
```

These aggregations provide a foundation for anomaly detection by highlighting deviations in summaries.

Advanced Time-Series Anomaly Detection with `series_decompose_anomalies()`

For time-series data in monitoring pipelines, `series_decompose_anomalies()` goes further by decomposing a series into baseline (expected), anomalies, and scores. It's built for sequential data like metrics over time.

- **Syntax:**
`series_decompose_anomalies(Series,
[Threshold, Seasonality, Trend,
Test_points, AD_method,
Seasonality_threshold])`
- **Series:** Numeric array (e.g., from `make-series`).
- **Threshold:** Anomaly sensitivity (default 1.5).
- **Seasonality:** Period detection (-1 for auto, or integer bins).
- **Trend:** 'avg', 'linefit', or 'none'.
- **Test_points:** Points to exclude for forecasting.

- **AD_method**: 'ctukey' or 'tukey' for outlier detection.
- **Seasonality_threshold**: Score threshold for auto-seasonality (default 0.6). Returns: Anomaly flags (+1/-1/0), scores, and baseline.

Building Custom Thresholds in Monitoring Pipelines

In pipelines, use this to create dynamic thresholds based on historical data. For login attempts:

```
let starttime = 7d;
```

```
let timeframe = 1h;
```

SigninLogs

```
| make-series Logins = count() default=0 on  
TimeGenerated from ago(starttime) to now() step  
timeframe
```

```
| extend (Anomalies, Score, Baseline) =  
series_decompose_anomalies(Logins, 2.0, -1,  
'linefit') // Custom threshold 2.0 for  
stricter detection  
  
| mv-expand Logins to typeof(double),  
TimeGenerated to typeof(datetime), Anomalies to  
typeof(double), Score to typeof(double),  
Baseline to typeof(long)  
  
| where Anomalies != 0  
  
| project TimeGenerated, Logins, Baseline,  
Score, Anomalies
```

Here, a threshold of 2.0 detects stronger anomalies, with 'linefit' for trend.

For network traffic volume, a comprehensive pipeline query:

```
let starttime = 14d;
```

```
let timeframe = 1h;
```

```
let scorethreshold = 5; // Custom score threshold

CommonSecurityLog

| where isnotempty(DestinationIP) and isnotempty(SourceIP)

| make-series Total=count() on TimeGenerated from ago(starttime) to now() step timeframe by DeviceVendor

| extend (Anomalies, Score, Baseline) = series_decompose_anomalies(Total, scorethreshold, -1, 'linefit')

| mv-expand Total to typeof(double), TimeGenerated to typeof(datetime), Anomalies to typeof(double), Score to typeof(double), Baseline to typeof(long)

| where Anomalies > 0 and Score > scorethreshold // Custom filtering
```

```
| project DeviceVendor, TimeGenerated, Total,  
Baseline, Anomalies, Score
```

This builds a monitoring flow: aggregate traffic, decompose for anomalies, and alert on high scores. Adjust scorethreshold based on your environment's noise level for custom thresholds.

By leveraging stdev(), variance(), percentiles_array(), and avgif(), you can uncover outliers in metrics like network traffic and login attempts that averages alone miss. For time-series data, series_decompose_anomalies() enables sophisticated decomposition and custom thresholding in automated pipelines, turning raw logs into proactive insights. Experiment with these in Azure Data Explorer or Sentinel to refine your detection strategies—start small, iterate on thresholds, and integrate with alerts for real impact.

Chapter 16

Query Optimization and Approximate Computations in KQL: Efficiency at Petabyte Scale

In the world of big data analytics, the Kusto Query Language (KQL) stands out as a powerful tool for querying massive datasets in Azure Data Explorer (ADX). Designed for handling telemetry, logs, and time-series data, KQL excels at processing petabytes of information with speed and scalability. However, when dealing with such volumes, query performance can become a bottleneck. Optimization techniques are essential to reduce latency, minimize resource consumption, and enable faster insights. This post dives into lesser-discussed aspects of KQL optimization: approximate computations using features like `hint.materialized` and sampling operators, as well as profiling with `set querytrace` and tuning partitioning policies. These methods are particularly valuable for exploratory analysis and production queries on enormous datasets.

Approximate Computations: Speeding Up Exploratory Queries

Exploratory data analysis often doesn't require exact results—approximations can provide quick insights without scanning every row in a petabyte-scale table. KQL offers built-in mechanisms to achieve this balance between accuracy and performance.

Sampling Operators for Quick Subsets

The sample operator is a straightforward way to retrieve a random subset of rows from a table, ideal for initial explorations where full precision isn't needed. It returns up to a specified number of rows, prioritizing speed over uniform distribution. For large datasets, apply it early in the query pipeline, right after table references or filters, to avoid processing unnecessary data.

For example, to get a quick sense of storm events without querying the entire dataset:

```
StormEvents
```

```
| where StartTime > ago(30d)  
  
| sample 1000  
  
| summarize Count=count() by State
```

This query samples 1,000 recent events and aggregates by state, running much faster than a full scan. For more structured sampling, use sample-distinct to select unique values from a column, then filter the original table:

```
let sampleStates = StormEvents | sample-  
distinct 10 of State;
```

StormEvents

```
| where State in (sampleStates)  
  
| summarize Events=count() by State
```

This approach is non-deterministic, meaning results vary per execution, but it's perfect for spotting trends in vast data without high compute costs. On petabyte scales, sampling reduces query time from minutes to seconds by limiting scanned extents.

Materialize for Caching and Consistent Approximations

For queries involving repeated subexpressions or non-deterministic functions, the `materialize()` function caches a tabular expression's results during query execution, allowing reuse without recomputation. This is especially useful for approximations, as it ensures consistency in functions like `dcount()` (approximate distinct count) or `rand()`.

Alternatively, use `hint.materialized=true` with operators like `as` or `partition` for similar caching. The cache is limited to 5 GB per node, so apply filters early to keep the materialized set small.

Here's an example caching a summarized dataset for multiple operations:

```
let cachedData = materialize(
```

StormEvents

```
| summarize Events=count() by State,  
EventType
```

) ;

cachedData

```
| summarize TotalEvents=sum(Events) by State
```

```
| join kind=inner (cachedData) on State
```

```
| extend Percentage = (Events * 100.0) / TotalEvents
```

| project State, EventType, Percentage

| top 10 by Percentage desc

By materializing once, this avoids redundant aggregations, cutting latency on large tables. For approximations, it pairs well with dcount():

```
let randomSample = materialize(range x from 1  
to 1000000 step 1 | project Value = rand());
```

```
randomSample | summarize ApproxUnique =  
dcount(Value)
```

This ensures dcount() uses the same cached data, providing a fast, approximate unique count without full scans—crucial for petabyte exploratory work.

Profiling Queries with set querytrace

To optimize, you first need visibility into query performance. The set querytrace option enables detailed tracing, revealing execution details like resource usage, stages, and bottlenecks. Set it to true before your query to get trace output alongside results.

For instance:

```
set querytrace = true;

StormEvents

| where DamageProperty > 1000000

| summarize TotalDamage=sum(DamageProperty) by
State

| top 5 by TotalDamage desc
```

The trace might show scanned extents, CPU time, and memory usage, helping identify inefficiencies like excessive shuffling or poor filtering. Use this for iterative tuning: run with trace, analyze, adjust (e.g., add filters earlier), and repeat. At petabyte scale, tracing uncovers why a query spikes resources, guiding optimizations like partitioning.

Tuning Partitioning for Reduced Latency

Data partitioning in ADX shards extents based on keys, enabling queries to skip irrelevant data and reduce latency. The partitioning policy controls this for tables or materialized views, but apply it judiciously—only in scenarios with frequent filters or aggregations on high-cardinality columns.

Key Types and Tuning

- **Hash Partitioning:** For string or GUID columns (e.g., TenantId), use XxHash64 with 128 partitions. Set PartitionAssignmentMode to Uniform for filters or ByPartition for joins/aggregations. This minimizes data movement in shuffle queries.
- **Uniform Range Datetime:** For time-based columns, partition by ranges (e.g., daily). Set OverrideCreationTime if data arrives out-of-order to align retention with actual timestamps.

To tune for latency:

- Start with MaxPartitionCount=128 to balance granularity and overhead.
- Monitor with .show table policy partitioning and adjust

EffectiveDateTime gradually for historical data to avoid resource spikes.

- Avoid if keys have skewed values (e.g., many nulls), as it can imbalance nodes.

Example policy for a tenant-based table:

```
.alter table MyTable policy partitioning  
@'{"PartitionKeys": [{"ColumnName": "TenantId",  
"Kind": "Hash", "Properties": {"Function":  
"XxHash64", "MaxPartitionCount": 128,  
"PartitionAssignmentMode": "Uniform"} } } '
```

Queries filtering on TenantId now scan fewer extents, slashing latency on petabytes. Combined with caching, this can reduce query times by orders of magnitude.

Mastering query optimization in KQL—through sampling and materialization for approximations, set querytrace for profiling, and partitioning tuning—unlocks efficient analysis of petabyte-scale data. These techniques, often underutilized, shift focus from brute-force scanning to smart, resource-aware querying. Experiment in your ADX environment, benchmark improvements, and scale your analytics confidently. For more,

explore the official KQL documentation and best practices.

Chapter 17

Mastering KQL Window Functions: Unlocking Sequential Analysis for Advanced Threat Hunting

Threat hunters don't just react to alerts—they proactively dissect timelines of events to uncover hidden narratives. Enter KQL (Kusto Query Language), the powerhouse query language behind Azure Data Explorer and Microsoft Sentinel. While most folks stick to the basics like summarize and where for sifting through logs, there's a lesser-known arsenal: **window functions**. These bad boys let you peer into sequences of events over sliding time windows, revealing patterns that scream "multi-stage attack" or "lateral movement."

If you've ever stared at a sea of timestamps wondering, "What *really* happened between that failed login and the admin privilege grab?", window functions are your crystal ball. They're rarely discussed beyond introductory tutorials, but they supercharge sequential analysis—think detecting brute-force attempts morphing into

privilege escalations or spotting anomalous session behaviors in real-time. In this post, we'll dive deep into key functions like `serialize`, `row_cumsum`, and `row_window_session`, with practical examples for threat hunting. We'll also layer in `prev` and `next` for delta magic on time-series threats. Buckle up; let's hunt some ghosts in the machine.

Why Window Functions Matter in Threat Hunting

Traditional queries aggregate data into neat buckets, but threats unfold *sequentially*. A single event might look innocent—a failed login here, a process spawn there—but string them over a time window, and boom: indicators of compromise (IoCs) emerge. Window functions operate on ordered rows, applying calculations like cumulative sums or session groupings without losing the raw timeline.

The payoff? Faster detection of:

- **Multi-stage attacks:** Recon → Exploitation → Persistence.

- **Behavioral anomalies:** Sudden spikes in session durations signaling data exfiltration.
- **Causal chains:** One event triggering the next, like a PowerShell script following a suspicious download.

Pro tip: Always start with extend or project to prep your data with timestamps, then feed it into these functions. Now, let's break 'em down.

The Building Blocks: Serialize for Order, Row_Cumsum for Momentum

Before windows can work their magic, your data needs to march in lockstep. That's where serialize comes in—it enforces a strict row order based on your specified column (usually a timestamp). Without it, KQL might shuffle rows unpredictably, turning your timeline into confetti.

Pair it with row_cumsum, which computes a running total over rows. Imagine tracking login failures: each flop increments a counter, and when it hits a threshold in a short window, you flag a brute-force blitz.

Example: Detecting Brute-Force Leading to Escalation

Let's say you're querying SigninLogs in Microsoft Sentinel. You want to spot repeated failures from the same IP, followed by a success that escalates privileges.

```
SigninLogs
```

```
| where TimeGenerated > ago(1d)
```

```
| where ResultType != 0    // Focus on failures first
```

```
| serialize by IPAddress, TimeGenerated    // Order by IP and time
```

```
| extend FailureCount = row_cumsum(1, IPAddress)    // Cumulative failures per IP
```

```
| where FailureCount > 5    // Threshold for  
suspicious activity  
  
| join kind=inner (SigninLogs  
  
| where ResultType == 0    // Successful  
logins  
  
| where TimeGenerated > ago(1d)  
  
| extend SuccessTime = TimeGenerated  
  
) on IPAddress  
  
| where SuccessTime between (TimeGenerated ..  
(TimeGenerated + 5m))    // Success within 5 mins  
of failures
```

```
| project TimeGenerated, IPAddress,  
FailureCount, AppDisplayName, UserPrincipalName  
  
| summarize by IPAddress
```

What this does:

- serialize lines up failures chronologically per IP.
- row_cumsum tallies them up—e.g., 7 failures in a row? Red flag.
- The join catches the "pivot" success, like a login that grants admin rights (check AppDisplayName for escalation hints).

Run this during an incident response, and you'll surface IPs attempting 10+ failures before slipping in via a weak password. In a real hunt, extend it to correlate with SecurityEvent logs for privilege use post-login.

Row_Window_Session: Grouping Events into Attack "Sessions"

Brute-force is child's play compared to sophisticated attacks that span minutes or hours. Enter row_window_session, which dynamically buckets rows into sessions based on a time gap. If

events are closer than your defined window (e.g., 10 minutes), they're one session; gaps reset the clock. Perfect for chaining multi-stage kills.

Example: Hunting Multi-Stage Attacks (Recon → Lateral Movement)

Picture an attacker: First, they probe ports (NetworkSecurityEvents), fail a few logins (SigninLogs), then escalate via a spawned process (SecurityEvent). Use row_window_session to group these into a single "attack session" if they occur within a 30-minute window.

```
let ReconEvents = NetworkSecurityEvents  
  
| where TimeGenerated > ago(1h)  
  
| where ActionType == "ConnectionDenied"    //  
Failed probes  
  
| summarize ProbeCount = count() by ClientIP,  
bin(TimeGenerated, 1m);
```

```
let LoginEvents = SigninLogs

| where TimeGenerated > ago(1h)

| where ResultType != 0

| summarize FailureCount = count() by
IPAddress, bin(TimeGenerated, 1m);

let EscalationEvents = SecurityEvent

| where TimeGenerated > ago(1h)

| where EventID == 4688 // Process creation
```

```
| where ProcessName contains "powershell.exe"  
or ProcessName contains "cmd.exe" //  
Suspicious spawns  
  
| summarize SpawnCount = count() by Computer,  
Account, bin(TimeGenerated, 1m);  
  
union ReconEvents, LoginEvents,  
EscalationEvents  
  
| serialize by ClientIP, IPAddress, Computer,  
TimeGenerated // Coalesce on actor identifiers  
  
| extend SessionId = row_window_session(30m,  
TimeGenerated, ClientIP) // 30-min session  
window per actor  
  
| where isnotempty(SessionId)
```

```
| summarize
```

```
TotalEvents = count(),
```

```
EventTypes =  
make_set($if(isempty(ProbeCount), "",  
"Probe")) , // Build pattern signature
```

```
EventTypes = strcat_array(EventTypes,  
$if(isempty(FailureCount), "", "LoginFail")),
```

```
EventTypes = strcat_array(EventTypes,  
$if(isempty(SpawnCount), "", "Escalation")),
```

```
StartTime = min(TimeGenerated),
```

```
EndTime = max(TimeGenerated)
```

```
by SessionId, ClientIP
```

```
| where array_length(split(EventTypes, ",")) >
2 // Multi-stage: At least 3 types

| project SessionId, ClientIP, EventTypes,
Duration = EndTime - StartTime
```

Breaking it down:

- union merges disparate logs into one stream.
- serialize + row_window_session groups into sessions—e.g., probes at T=0, logins at T=5m, spawns at T=20m? One cohesive attack session.
- The summarize crafts a "pattern fingerprint" via make_set and strcat_array. Spot "Probe,LoginFail,Escalation"? Escalate to your IR team.

This query shines in proactive hunts: Pipe it into a workbook for visualization, and you'll see attack timelines as neat clusters. Tweak the window to 1h for longer dwell times.

Leveling Up with Prev/Next: Delta Calculations for Time-Series Threats

Windows are great for aggregates, but what about *changes* between events? prev and next let you peek at adjacent rows in your serialized stream, enabling delta calcs like session duration or velocity (events per minute).

Example: Anomalous Session Durations in Time-Series Threats

Exfiltration often hides in prolonged sessions. Track prev session end to compute deltas, flagging outliers.

IdentityInfo

```
| join kind=inner (
```

SigninLogs

```
| where TimeGenerated > ago(1d)
```

```
| where ResultType == 0
```

```
| serialize by UserPrincipalName,  
TimeGenerated  
  
| extend PrevSignIn = prev(TimeGenerated,  
1, UserPrincipalName),  
  
SessionDuration = TimeGenerated -  
PrevSignIn  
  
| where isnotempty(PrevSignIn)  
  
| extend IsAnomalous = SessionDuration > 1h  
// Flag sessions >1 hour  
  
| summarize AvgDuration =  
avg(SessionDuration), MaxDuration =  
max(SessionDuration),  
  
AnomalousSessions =  
countif(IsAnomalous)
```

```
    by UserPrincipalName,  
bin(TimeGenerated, 1h)  
  
) on UserPrincipalName  
  
  
| where MaxDuration > 30m // Focus on  
prolonged activity  
  
  
| extend RiskScore = AnomalousSessions *  
(MaxDuration / 1h) // Simple scoring  
  
  
  
| project UserPrincipalName, AvgDuration,  
MaxDuration, RiskScore  
  
  
  
| order by RiskScore desc
```

The secret sauce:

- serialize ensures chronological order per user.
- prev(TimeGenerated, 1) grabs the last sign-in time, so SessionDuration is the gap.

- Combine with `row_cumsum` (from earlier) to track cumulative anomalous sessions over a window.

This catches insiders or compromised accounts lingering too long—pair with `next` to preview the *next* action (e.g., data access post-long session).

From Sequences to Security Wins

KQL's window functions aren't just syntax sugar; they're the thread that weaves raw logs into threat stories. By mastering `serialize` for order, `row_cumsum` for tallies, `row_window_session` for grouping, and `prev/next` for dynamics, you'll detect multi-stage attacks before they pivot and spot time-series oddities that AV misses.

Next time you're in Sentinel, ditch the point-and-click—query like a hunter. Experiment with these in a test workspace, and share your tweaks. What's your go-to window function for hunting? Stay vigilant out there.

Happy querying, and may your timelines always tell tales worth chasing.

Chapter 18

Simulating Graph-Like Traversals and Recursive Patterns in KQL

Kusto Query Language (KQL), the powerhouse behind Azure Data Explorer, Microsoft Sentinel, and other Azure services, is renowned for its speed and efficiency in handling massive datasets. However, one area where it has traditionally fallen short is native support for graph databases—those interconnected webs of nodes and edges perfect for modeling relationships like social networks, hierarchies, or permissions. While recent updates have introduced graph semantics (like the make-graph operator), simulating graph traversals in KQL often feels like a hacky adventure, relying on clever workarounds such as recursive unions or array manipulations. This makes it a niche, under-discussed topic, but one that's incredibly useful for scenarios like security analysis or network topology.

In this post, we'll dive into two key approaches: using recursive unions (and the newer make-graph) for traversing relationships, such as user-

group-permission chains, and approximating pathfinding with series functions for network analysis. We'll include practical examples to get you started. Note that while graph semantics are now built-in, we'll emphasize the "hacky" simulation methods for those working in environments without full access to the latest features or preferring custom tweaks.

Recursive Unions for Relationship Traversal

Before graph semantics became a thing, KQL users simulated recursive traversals by building "levels" of a hierarchy through repeated joins and unions. This isn't true recursion (KQL doesn't support recursive functions directly), but it approximates depth-limited tree or graph walking by iteratively expanding layers. It's perfect for chains like user → group → permission, where you might need to trace access rights in an identity system.

The basic idea: Start with a seed (e.g., a user), join to find direct connections (e.g., groups), union the results, and repeat for the next level (e.g., permissions). You define a fixed depth to avoid

infinite loops—say, up to 5-10 levels, depending on your data.

Example: Tracing User-Group-Permission Chains

Imagine you have a table `Identities` with columns for `EntityId`, `ParentId` (linking users to groups or groups to permissions), `Type` (`user/group/permission`), and `Name`. To build a flat tree starting from a user "Alice" and trace her effective permissions:

```
let Identities = datatable(EntityId: string,  
    ParentId: string, Type: string, Name: string)
```

```
[
```

```
"U1", "G1", "user", "Alice",
```

```
"G1", "P1", "group", "Admins",
```

```
"P1", "", "permission", "ReadWriteAll",
"U2", "G2", "user", "Bob",
"G2", "P2", "group", "Readers",
"P2", "", "permission", "ReadOnly"
];
let Level0 = Identities | where Name == "Alice"
| project EntityId, Path = pack_array(Name),
Depth = 0;
let Level1 = Level0
| join kind=inner (Identities) on
$left.EntityId == $right.ParentId
```

```
| project EntityId = EntityId1, Path =
array_concat(Path, Name1), Depth = Depth + 1;

let Level2 = Level1

| join kind=inner (Identities) on
$left.EntityId == $right.ParentId

| project EntityId = EntityId1, Path =
array_concat(Path, Name1), Depth = Depth + 1;

Level0 | union Level1 | union Level2

| where Type == "permission"    // Filter to end
at permissions

| project Path, Depth
```

This query outputs something like:

Here, we've "recursively" traversed two levels to find Alice's permissions via her group. For deeper chains, add more levels (e.g., Level3, Level4). It's hacky but effective for bounded depths. If your hierarchy is unpredictable, this can get verbose—union up to a reasonable max depth and filter out incomplete paths.

In security contexts, like Azure AD logs, this pattern helps uncover privilege escalations by chaining user-group-role assignments from tables like SigninLogs or custom identity data.

Using `make_graph` for More Structured Traversals

If you're on a recent KQL version (post-2024 updates), the `make-graph` operator lets you build transient graphs directly, making traversals less hacky. It takes edges (relationships) and optional nodes (entities), then pairs with `graph-match` for pattern matching. This is ideal for the same user-group-permission scenarios but with variable-depth paths using `*` (Kleene star) for repetition.

Syntax Basics

- **Edges:** A table with source and target columns (e.g., UserId → GroupId).
- **Nodes:** Optional tables with entity details.
- Follow with graph-match to query patterns like (user)-[membership*1..5]->(permission).

Example: User-Group-Permission Chains with make_graph

Using similar data:

```
let Nodes = datatable(NodeId: string, Type: string, Name: string)
```

```
[
```

```
"U1", "user", "Alice",
```

```
"G1", "group", "Admins",
```

```
"P1", "permission", "ReadWriteAll"  
]  
  
let Edges = datatable(Source: string, Target: string, Relationship: string)  
[  
  
    "U1", "G1", "memberOf",  
  
    "G1", "P1", "hasPermission"  
]  
  
Edges  
  
| make-graph Source --> Target with Nodes on  
NodeId
```

```
| graph-match (user)-[rel*1..3]->(perm)  
  
where user.Type == "user" and perm.Type ==  
"permission"  
  
and rel.Relationship in ("memberOf",  
"hasPermission")  
  
| project User = user.Name, Path =  
array_concat(pack_array(user.Name),  
rel.Relationship, pack_array(perm.Name))
```

This finds paths like Alice → memberOf → Admins → hasPermission → ReadWriteAll. The *1..3 allows variable-length traversals up to depth 3, simulating recursion elegantly. For complex permissions in cloud environments, model users/groups as nodes and memberships/roles as edges—great for spotting over-privileged paths in resource hierarchies.

In security hunting, this shines for lateral movement: Query paths like (compromisedUser)-[adminTo*3..9]->(domainAdmin) to trace attack vectors in logs from Microsoft Defender.

Pathfinding Approximations with Series Functions for Network Topology Analysis

For network topology—think device connections, traffic flows, or IoT meshes—true pathfinding (e.g., shortest path) isn't native in KQL without graphs. Here, we hack it with dynamic arrays (treated as series) to represent paths, building them recursively via unions, then using series functions for approximations like path length stats or anomaly detection in topologies.

Series functions (e.g., `series_stats`, `series_fit_line`) typically handle time series, but we repurpose them for array-based "paths" (e.g., `[hop1, hop2, hop3]`). This approximates pathfinding by iteratively expanding possible paths and analyzing the resulting series for metrics like average hops or outliers in network latency.

Example: Approximating Paths in a Network Topology

Suppose a `NetworkLogs` table with `SourceIP`, `TargetIP`, `HopCount`, and `Latency`. To approximate paths from a starting IP and analyze topology:

```
let NetworkLogs = datatable(SourceIP: string,  
TargetIP: string, HopCount: int, Latency:  
double)
```

```
[
```

```
    "192.168.1.1", "192.168.1.2", 1, 5.0,
```

```
    "192.168.1.2", "192.168.1.3", 1, 3.0,
```

```
    "192.168.1.1", "192.168.1.3", 2, 8.0    //  
Direct longer path
```

```
];
```

```
let StartIP = "192.168.1.1";
```

```
let Level0 = NetworkLogs | where SourceIP ==  
StartIP | project Path = pack_array(SourceIP,  
TargetIP), TotalHops = HopCount, TotalLatency =  
Latency;
```

```
let Level1 = Level0

| join kind=inner (NetworkLogs) on
$left.TargetIP == $right.SourceIP

| where not(array_index_of(Path, TargetIP1)
>= 0) // Avoid cycles

| project Path = array_concat(Path,
TargetIP1), TotalHops = TotalHops + HopCount1,
TotalLatency = TotalLatency + Latency1;

Level0 | union Level1

| summarize Paths = make_list(Path), HopSeries
= make_list(TotalHops), LatencySeries =
make_list(TotalLatency) by StartIP

| extend HopStats = series_stats(HopSeries),
LatencyFit = series_fit_line(LatencySeries)
```

```
| project StartIP, AvgHops = HopStats.avg,  
ShortestPath =  
Paths[series_arg_min(HopSeries)[0]],  
LatencySlope = LatencyFit.slope
```

This builds path arrays recursively (limited depth), then uses series_stats for hop averages and series_fit_line to approximate latency trends across paths—handy for spotting inefficient topologies or anomalies in network simulations. In real network analysis, extend this to filter shortest paths or detect bottlenecks in topologies like mesh networks.

Simulating graphs in KQL might feel makeshift, but techniques like recursive unions and series hacks unlock powerful traversals for relationships and topologies. With the advent of make-graph, things are getting smoother, but the old-school methods remain valuable for custom needs. Experiment in your Azure Data Explorer cluster—start small with sample data, and scale to real logs. If you're dealing with security or networks, these patterns can reveal hidden insights.

Chapter 19

Beyond the Basics: Unlocking the Power of KQL Window Functions

While basic KQL queries handle filtering, grouping, and joins effectively, window functions take data analysis to the next level. These functions enable row-wise operations, ranking, moving averages, and advanced aggregations without collapsing the dataset into summarized rows. In this post, we'll explore the power of KQL window functions with practical examples to demonstrate their versatility.

What Are Window Functions in KQL?

Window functions in KQL perform calculations across a set of rows (a "window") related to the current row, without grouping the entire dataset. Unlike traditional aggregations (e.g., `summarize`), window functions preserve the original rows, adding computed columns for each row based on its window. This makes them ideal for tasks like

ranking, running totals, moving averages, and comparing rows within a partition.

Key characteristics of KQL window functions:

- Operate over a defined window of rows, often partitioned by a column and ordered by another.
- Support functions like `row_number()`, `rank()`, `sum()`, `avg()`, and more.
- Use the `over` clause to define the window's scope.

Let's dive into practical examples to see window functions in action.

Row-Wise Operations

with `row_number()` and `prev()`

Window functions shine in scenarios where you need to assign a unique number to each row or access values from previous rows.

Example: Assigning Row Numbers

Suppose you have a dataset of user login events and want to number each login per user chronologically.

```
let Logins = datatable(UserId: string,
LoginTime: datetime)

[
    "Alice", datetime(2025-09-01 10:00:00),
    "Alice", datetime(2025-09-01 12:00:00),
    "Bob",   datetime(2025-09-01 09:00:00),
    "Bob",   datetime(2025-09-01 11:00:00)
];

Logins
| order by UserId asc, LoginTime asc
```

```
| extend RowNum = row_number() over (partition  
by UserId order by LoginTime)
```

Output:

UserId	LoginTime	RowNum
Alice	2025-09-01 10:00:00	1
Alice	2025-09-01 12:00:00	2
Bob	2025-09-01 09:00:00	1
Bob	2025-09-01 11:00:00	2

Here, `row_number()` assigns a sequential number to each login within each `UserId` partition, ordered by `LoginTime`.

Example: Accessing Previous Row Values

To calculate the time difference between consecutive logins for each user, use the `prev()` function.

Logins

```
| order by UserId asc, LoginTime asc  
  
| extend PrevLoginTime = prev(LoginTime, 1)  
over (partition by UserId)  
  
| extend TimeDiff = LoginTime - PrevLoginTime
```

Output:

UserId	LoginTime	PrevLoginTime	TimeDiff
Alice	2025-09-01 10:00:00		
Alice	2025-09-01 12:00:00	2025-09-01 10:00:00	02:00:00
Bob	2025-09-01 09:00:00		
Bob	2025-09-01 11:00:00	2025-09-01 09:00:00	02:00:00

The `prev()` function retrieves the `LoginTime` of the previous row within the same `UserId` partition, enabling time difference calculations.

Ranking with `rank()` and `dense_rank()`

Ranking functions assign positions to rows based on a specified order, useful for leaderboards or identifying top performers.

Example: Ranking Sales by Region

Imagine a sales dataset where you want to rank salespeople within each region by their total sales.

```
let Sales = datatable(Region: string,  
Salesperson: string, Amount: double)
```

```
[
```

```
    "East", "Alice", 500,
```

```
    "East", "Bob", 700,
```

```
    "West", "Charlie", 600,
```

```
"West", "Dave", 600
```

```
];
```

```
Sales
```

```
| order by Region asc, Amount desc
```

```
| extend Rank = rank() over (partition by  
Region order by Amount desc),
```

```
DenseRank = dense_rank() over  
(partition by Region order by Amount desc)
```

Output:

Region	Salesperson	Amount	Rank	DenseRank
East	Bob	700	1	1
East	Alice	500	2	2
West	Charlie	600	1	1
West	Dave	600	1	1

- `rank()` assigns the same rank to tied values (e.g., Charlie and Dave both get rank 1) and skips subsequent ranks (e.g., no rank 2 in West).
- `dense_rank()` assigns the same rank to ties but does not skip ranks (e.g., the next rank after 1 is 2).

Moving Averages with `avg()`

Window functions make it easy to compute moving averages, which smooth out data trends over a sliding window.

Example: Calculating a 3-Day Moving Average

For a dataset of daily stock prices, compute a 3-day moving average for each stock.

```
let StockPrices = datatable(Stock: string,  
Date: datetime, Price: double)
```

```
[
```

```
"AAPL", datetime(2025-09-01), 150,
```

```
"AAPL", datetime(2025-09-02), 152,
```

```
"AAPL", datetime(2025-09-03), 155,
```

```
"AAPL", datetime(2025-09-04), 157,
```

```
"MSFT", datetime(2025-09-01), 300,
```

```
"MSFT", datetime(2025-09-02), 305
```

```
] ;
```

```
StockPrices
```

```
| order by Stock asc, Date asc
```

```
| extend MovingAvg = avg(Price) over (partition  
by Stock order by Date rows between 2 preceding  
and current row)
```

Output:

Stock	Date	Price	MovingAvg
AAPL	2025-09-01	150	150.00
AAPL	2025-09-02	152	151.00
AAPL	2025-09-03	155	152.33
AAPL	2025-09-04	157	154.67
MSFT	2025-09-01	300	300.00
MSFT	2025-09-02	305	302.50

The `rows between 2 preceding and current row` clause defines a window of the current row and the two previous rows, computing the average price within that window for each stock.

Advanced Aggregations

with `sum()` and `count()`

Window functions can compute running totals or cumulative counts, preserving the dataset's granularity.

Example: Running Total of Sales

Calculate a running total of sales amounts for each region.

```
Sales
```

```
| order by Region asc, Amount desc
```

```
| extend RunningTotal = sum(Amount) over  
(partition by Region order by Amount desc)
```

Output:

Region	Salesperson	Amount	Running Total
East	Bob	700	700
East	Alice	500	1200
West	Charlie	600	600
West	Dave	600	1200

The `sum()` function accumulates the `Amount` within each `Region` partition, ordered by `Amount` descending.

Best Practices and Tips

- 1. Partition Wisely:** Use `partition by` to group related rows (e.g., by user or region) to ensure calculations are contextually relevant.
- 2. Order Matters:** The `order by` clause in the `over` statement determines the sequence of rows in the window, critical for functions like `row_number()` or `sum()`.
- 3. Performance Considerations:** Window functions can be resource-intensive on large datasets. Use them judiciously and test performance on representative data.

4. Combine with Other KQL Features: Pair window functions with `where`, `join`, or `summarize` for more complex analyses.

KQL window functions unlock powerful analytical capabilities, allowing you to perform row-wise operations, rankings, moving averages, and advanced aggregations while preserving your dataset's structure. By mastering functions like `row_number()`, `rank()`, `avg()`, and `sum()` with the `over` clause, you can tackle sophisticated data analysis tasks with ease. Experiment with these examples in Azure Data Explorer to see how window functions can transform your queries and reveal deeper insights.

Chapter 20

Visualizing KQL: Turning Queries into Stunning Dashboards

In the world of big data and analytics, extracting insights from vast datasets is only half the battle. The real power lies in visualizing those insights to make them actionable and understandable.

Enter Kusto Query Language (KQL), a robust query language designed for querying structured, semi-structured, and unstructured data in Azure Data Explorer (ADX), Azure Monitor, and other Microsoft services. KQL excels at handling telemetry, logs, and metrics, making it ideal for time-series analysis, aggregations, and pattern detection.

This chapter will guide you through using KQL to extract and summarize data, then transform it into stunning dashboards using tools like Azure Data Explorer's built-in dashboards and Power BI. We'll include a hands-on tutorial with query examples, step-by-step instructions, and tips for creating compelling visuals. Whether you're a data analyst, engineer, or enthusiast, by the end,

you'll be equipped to turn raw queries into insightful visualizations.

Getting Started with KQL: Extracting and Summarizing Data

KQL is read-only, expressive, and optimized for data exploration. Queries follow a data-flow model, where you pipe operators (using |) to filter, transform, and summarize data. Let's start with the basics using the sample `StormEvents` table (a common dataset in ADX tutorials for weather events).

Tutorial: Writing Queries to Summarize Data

Filtering Data with `where`: Before summarizing, filter your data to focus on relevant subsets. For example, to get storm events in Florida during November 2007:

```
StormEvents
```

```
| where StartTime between  
(datetime(2007-11-01) .. datetime(2007-  
12-01))  
  
| where State == "FLORIDA"
```

This returns rows matching the conditions. Visual tip: This filtered data can feed into a table visual for quick inspection.

Aggregating with summarize and count: To count events:

StormEvents

```
| where StartTime between  
(datetime(2007-11-01) .. datetime(2007-  
12-01))  
  
| where State == "FLORIDA"  
  
| count
```

Result: A single value, e.g., Count = 123.
For grouping by state:

StormEvents

```
| summarize EventCount = count() by  
State  
  
| project State, EventCount
```

This groups events by state and counts them, then uses project to select and rename columns. Output: A table like:

State	EventCount
TEXAS	500
FLORIDA	123
...	...

Time-Series Summarization with bin:
For temporal data, bin time into intervals (e.g., daily):

StormEvents

```
| where StartTime >= datetime(2007-11-01) and StartTime <= datetime(2007-12-01)

| summarize DailyCount = count() by bin(StartTime, 1d)

| project Date = StartTime, DailyCount
```

This aggregates events into daily buckets. Result: A table showing counts per day, perfect for trend analysis.

These queries demonstrate core operators: where for filtering, summarize for aggregation (with functions like count(), avg(), sum()), bin for time grouping, and project for column selection. Experiment in the ADX web UI for instant results.

Visualizing Data Directly in KQL with the render Operator

KQL isn't just for querying— it has built-in visualization capabilities via the render operator.

Append it to your query to turn tabular results into charts without leaving ADX.

Examples:

Bar Chart for Event Counts by State:

```
StormEvents
```

```
| summarize EventCount = count() by State
```

```
| render barchart with (title="Storm Events by State")
```

This renders a bar chart where states are on the x-axis and counts on the y-axis. Imagine vertical bars, with Texas towering high if it has the most events—great for comparing categories.

Time Chart for Daily Trends:

```
StormEvents
```

```
| where StartTime >= datetime(2007-01-01) and StartTime <= datetime(2007-12-31)
```

```
| summarize DailyCount = count() by bin(StartTime, 1d)
```

```
| render timechart with (title="Daily Storm Events in 2007", xtitle="Date", ytitle="Count")
```

Visual: A line chart showing fluctuations over time, with peaks during storm seasons. Use anomalychart for detecting outliers.

Pie Chart for Proportions:

StormEvents

```
| summarize count() by EventType
```

```
| render piechart with (title="Storm Event Types")
```

This creates a pie chart slicing data by event types (e.g., Thunderstorm, Flood). Ideal for showing parts of a whole.

The render operator supports types like columnchart, scatterchart, areachart, and more. Customize with options for titles, axes, and legends.

Building Dashboards in Azure Data Explorer

ADX dashboards let you compile multiple KQL-based visuals into interactive pages.

Step-by-Step Tutorial:

1. Create a New Dashboard:

- In the ADX web UI, go to **Dashboards > New dashboard**.
- Name it (e.g., "Storm Analytics Dashboard") and create.

2. Add a Data Source:

- Select **More [...] > Data sources > + New data source**.

- Enter name, cluster URI, database, and cache age.
Connect to your ADX cluster.

3. Pin Queries as Tiles:

- Run a query in the query editor (e.g., the bar chart above).
- Select **Share > Pin to dashboard**.
- Choose tile name, data source, and dashboard. Pin it.
- Alternatively, add a tile directly: **Add tile**, select data source, write KQL, run, choose visual type (e.g., Bar), and apply.

4. Customize and Add Parameters:

- Resize tiles, edit visuals (e.g., change legend position).
- Add parameters for dynamic filtering: Define in dashboard settings, use in queries like | where State == \$stateParams.
- Organize into pages: **+ Add page** for grouping related visuals.

5. Enable Auto-Refresh:

- Set intervals (e.g., every 5 minutes) for real-time updates.

Example Dashboard Visual: Picture a dashboard with a timechart tile showing daily trends on top, a pie chart of event types below, and a bar chart of counts by state on the side—all interactive, with hover tooltips and filters.

Integrating KQL with Power BI for Advanced Dashboards

Power BI takes visualization further with rich interactivity and sharing.

Step-by-Step Tutorial:

1. Connect Power BI to ADX:

- In Power BI Desktop, go to **Get Data > More > Azure > Azure Data Explorer (Kusto)**.
- Enter cluster URL (e.g.,

<https://help.kusto.windows.net>

1.

-), database name, and sign in with Azure AD credentials.
- Use DirectQuery for real-time data or Import for snapshots.

2. Import Data with KQL:

- In the navigator, write or paste a KQL query (e.g., the summarization by state).
- Load the data. Power BI converts it to a table.

3. Create Visuals:

- Drag fields to visuals: Create a clustered bar chart for states vs. counts.
- Add slicers for filters (e.g., date range).
- Build a report page with multiple visuals, like a map if geospatial data is available.

4. Publish and Share:

- Publish to Power BI service for dashboards and sharing.

Best Practice: Use DirectQuery for large datasets to avoid import limits, and optimize queries for performance.

Example Visual in Power BI: Envision a dynamic dashboard with a line chart tracking events over time, synced filters, and drill-down capabilities—far more engaging than static charts.

Tips for Creating Compelling Visuals

- **Choose the Right Chart Type:** Use timecharts for trends, bar/pie for categories, scatter for correlations. Avoid clutter—simpler is better.
- **Interactivity Matters:** Add parameters, legends, and tooltips for exploration. In Power BI, use slicers and cross-filtering.
- **Performance Optimization:** Use aggregations early in queries to reduce data volume. Set cache in ADX dashboards.
- **Design Principles:** Consistent colors, clear titles, and logical layouts. Group related visuals; use pages in ADX for organization.
- **Storytelling:** Combine visuals to tell a story—e.g., overview chart + detailed breakdowns.
- **Test and Iterate:** Run queries on sample data first; preview visuals to ensure they convey the intended message.

KQL empowers you to go from raw data to insightful dashboards seamlessly. By mastering summarization queries and leveraging tools like ADX dashboards and Power BI, you can create visuals that drive decisions. Start with the examples here, experiment in your environment,

and watch your data come alive. For more advanced topics, explore Microsoft docs on KQL aggregations and visualizations. If you'd like me to generate custom images for these visuals (e.g., sample charts), confirm and provide details!

Chapter 21

Unpacking the Magic: A Fun Dive into KQL's `bag_unpack` Operator!

Hey there, data explorers! Imagine you're a wizard in the world of Azure Data Explorer, and you've got this enchanted bag full of goodies—mysterious properties hiding inside a dynamic column. But oh no, you can't query them easily because they're all bundled up! Enter the **bag_unpack** operator, your magical spell to unzip that bag and lay out all the treasures as neat, queryable columns. It's like turning a messy suitcase into an organized wardrobe. Fun, right? Today, we're going on a whimsical adventure to learn how to use it. We'll keep it light, sprinkle in some examples, and by the end, you'll be unpacking bags like a pro!

What on Earth is `bag_unpack`?

In Kusto Query Language (KQL), data often comes in "dynamic" types—think JSON-like objects called property bags. These are flexible but can be a pain when you want to treat their keys as

columns for filtering, aggregating, or joining. The `bag_unpack` plugin (yep, it's a plugin invoked via the `evaluate` operator) takes that dynamic column and explodes it into individual columns, one for each top-level property. It's the opposite of stuffing things into a bag; it's *unstuffing* them!

Why bother? Well, imagine analyzing logs where each event has a bag of attributes like `{"Color": "Red", "Size": 42}`. Without unpacking, you'd be poking around with dot notation or functions like `bag_keys()`. But unpack it, and boom—now you have "Color" and "Size" as columns you can sort, filter, or visualize directly. Efficiency level: Expert wizard!

The Spellbook: Syntax and Parameters

Casting this spell is straightforward. The basic syntax looks like this:

```
T | evaluate bag_unpack(Column [,  
OutputColumnPrefix] [, columnsConflict] [,  
ignoredProperties]) [: OutputSchema]
```

- **T**: Your input table (the one with the magical bag).

- **Column**: The dynamic column you want to unpack (required!).
- **OutputColumnPrefix**: Optional string to prefix all new columns (handy to avoid name clashes, like adding "Magic_" to everything).
- **columnsConflict**: How to handle if unpacked columns clash with existing ones. Options: 'error' (default, throws a tantrum), 'replace_source' (overwrite the old with the new), or 'keep_source' (keep the old, ignore the new).
- **ignoredProperties**: A dynamic array of properties to skip unpacking (for when some stuff in the bag is junk).
- **OutputSchema**: Optional but super important—defines the output columns and types explicitly, like (Age: long, Name: string). Use a wildcard * to include all original columns: (*, Age: long, Name: string). Pro tip: Always use this for big data to avoid performance hiccups!

Bag_unpack returns a table with the same number of rows as your input, but with the dynamic column removed and new columns added for each unique property. If properties have

mixed types across rows, they become dynamic.
Null bags? Ignored like yesterday's socks.

Performance Potion: Skip the OutputSchema on huge datasets at your peril—it forces KQL to scan everything to figure out the schema, which can be sloooow. Define it upfront for speedy queries!

Let's Get Hands-On: Fun Examples!

Time to play! We'll use a simple datatable of wizard apprentices with their stats in a bag. (In real life, this could be IoT data, logs, or anything nested.)

Example 1: Basic Unpacking – The Simple Spell

Suppose we have this table:

```
datatable(d: dynamic)
```

```
[
```

```
dynamic({"Name": "Gandalf", "Age": 2000,  
"PowerLevel": 99}),  
  
dynamic({"Name": "Hermione", "Age": 17,  
"PowerLevel": 85}),  
  
dynamic({"Name": "Dumbledore", "Age": 150,  
"PowerLevel": 100})  
]  
  
| evaluate bag_unpack(d)
```

What happens? The bag bursts open!

See? Each key becomes a column. Now you can easily query where PowerLevel > 90 to find the top wizards!

Example 2: Prefix Power – Avoiding Mix-Ups

If your table already has a "Name" column (maybe from elsewhere), add a prefix to keep things tidy:

```
datatable(d: dynamic)
```

```
[
```

```
    dynamic({ "Name": "Gandalf", "Age": 2000,  
    "PowerLevel": 99}),
```

```
    dynamic({ "Name": "Hermione", "Age": 17,  
    "PowerLevel": 85}),
```

```
    dynamic({ "Name": "Dumbledore", "Age": 150,  
    "PowerLevel": 100})
```

```
]
```

```
| evaluate bag_unpack(d, 'Wizard_')
```

Output:

Wizard_Age	Wizard_Name	Wizard_PowerLevel
2000	Gandalf	99
17	Hermione	85
150	Dumbledore	100

No collisions, no drama. Like labeling your potion bottles!

Example 3: Conflict Resolution – The Drama Queen

What if there's already a "Name" column? Let's say our table has an old, boring "Name" like "Unknown Wizard":

```
datatable(Name: string, d: dynamic)
```

```
[
```

```

    "Unknown Wizard", dynamic({"Name": "Gandalf",
    "Age": 2000}),

    "Unknown Wizard", dynamic({"Name": "Hermione",
    "Age": 17}),

    "Unknown Wizard", dynamic({"Name": "Dumbledore",
    "Age": 150})

]

| evaluate bag_unpack(d,
columnsConflict='replace_source')

```

With 'replace_source', the unpacked "Name" overwrites the old one:

Age	Name
2000	Gandalf
17	Hermione
150	Dumbledore

Flip to 'keep_source' to preserve the original:

Age	Name
2000	Unknown Wizard
17	Unknown Wizard
150	Unknown Wizard

Example 4: Ignoring Junk and Defining Schema – Advanced Wizardry

Got some properties you don't care about? Ignore 'em! And always define that schema for speed:

```
datatable(d: dynamic)
```

```
[
```

```
    dynamic({ "Name": "Gandalf", "Age": 2000,  
    "PowerLevel": 99, "Secret": "Shhh" }),
```

```
    dynamic({ "Name": "Hermione", "Age": 17,  
    "PowerLevel": 85, "Secret": "Books" })
```

```
]
```

```
| evaluate bag_unpack(d,  
ignoredProperties=dynamic(["Secret"])) : (Name:  
string, Age: long, PowerLevel: int)
```

Output (no "Secret" column!):

Hermione	17	82
Ginny	1500	66
Seamus	AgeA	largePower

By specifying the schema, KQL doesn't have to guess types or scan deeply—your query flies!

Tips and Tricks from the Wizard's Hat

- **Combine with Others:** Use `bag_unpack` after `mv-expand` for arrays inside bags, or with `make_bag` to reverse the process.
- **Watch for Dynamics:** If properties vary wildly, columns might end up as dynamic—cast them in the schema for consistency.
- **Large Data? Schema Up!:** As mentioned, `OutputSchema` is your best friend for performance.
- **Limitations:** It only unpacks top-level properties (no deep nesting here—use

mv-expand for arrays). And output names must be valid identifiers.

- **Fun Fact:** Think of it as KQL's way of saying, "Let me help you declutter your data attic!"

Wrapping Up the Bag

There you have it, fellow queriers—the bag_unpack operator demystified and defunned! It's a simple yet powerful tool to flatten your dynamic data and make queries sparkle. Next time you're wrestling with property bags, just whisper "evaluate bag_unpack" and watch the magic happen.

Chapter 22

Powerful KQL Operators You Didn't Know You Needed

The Kusto Query Language (KQL) is a robust tool for querying and analyzing data in Azure Data Explorer, Log Analytics, and other Microsoft services. While most users are familiar with basic operators like `where`, `summarize`, and `join`, KQL offers a treasure trove of lesser-known operators that can supercharge your data analysis. In this chapter, we'll explore five powerful KQL operators that you might not have realized you needed, complete with practical examples to demonstrate their utility.

1. `mv-expand`: Unraveling Arrays and Dynamic Types

The `mv-expand` operator is a game-changer when dealing with dynamic data types, such as arrays or JSON objects. It expands multi-valued columns into individual rows, making it easier to analyze nested or complex data structures.

Example: Analyzing Log Data with Arrays

Suppose you have a table `Logs` with a column `Tags` containing arrays of tags associated with each log entry. You want to analyze the frequency of each tag.

```
Logs
```

```
| where Timestamp > ago(7d)
```

```
| mv-expand Tags
```

```
| summarize TagCount = count() by  
tostring(Tags)
```

```
| order by TagCount desc
```

What it does: This query takes the `Tags` array, expands it so each tag gets its own row, and then counts the occurrences of each tag.

The `tostring` function ensures the tag is treated as

a string for grouping. This is invaluable for log analysis where tags or labels are stored as arrays.

Why you need it: Without `mv-expand`, you'd struggle to process array data without complex parsing or scripting. It simplifies working with JSON-like structures, which are common in telemetry and log data.

2. `make-series`: Time Series Analysis Made Simple

The `make-series` operator creates a time series from your data, aggregating values over regular time intervals. It's perfect for trend analysis, especially when dealing with time-based data like metrics or logs.

Example: Tracking CPU Usage Over Time

Imagine a `Metrics` table with CPU usage data. You want to visualize average CPU usage per hour over the last day.

Metrics

```
| where Timestamp > ago(1d)

| make-series AvgCPU = avg(CPUUsage) default=0
on Timestamp from ago(1d) to now() step 1h by
ServerName

| project ServerName, AvgCPU, Timestamp
```

What it does: This query creates a time series of average CPU usage (`AvgCPU`) for each `ServerName`, with data points every hour. The `default=0` ensures missing intervals are filled with zeros, and the `project` operator formats the output for clarity.

Why you need it: `make-series` is essential for creating smooth, continuous time series data for charting or anomaly detection, saving you from manual bucketing or scripting.

3. `parse`: Extracting Data from Strings

The `parse` operator is a powerful tool for extracting structured data from unstructured or semi-structured strings, such as log messages or URLs. It allows you to define a pattern and map parts of the string to new columns.

Example: Parsing URLs in Web Logs

Suppose you have a `WebLogs` table with a `RequestUrl` column containing URLs like `https://example.com/api/v1/users/123`. You want to extract the endpoint and user ID.

`WebLogs`

```
| parse RequestUrl with * "/api/v1/" Endpoint  
"/" UserId
```

```
| summarize RequestCount = count() by Endpoint,  
UserId
```

What it does: The `parse` operator matches the URL pattern, extracting the `Endpoint` (e.g., `users`) and `UserId` (e.g., `123`) into new columns.

The `summarize` operator then counts requests by endpoint and user.

Why you need it: Parsing unstructured data like logs or URLs is a common challenge.

The `parse` operator simplifies this without requiring complex regular expressions or external tools.

4. `top-nested`: Hierarchical Aggregation

The `top-nested` operator is a hidden gem for hierarchical data analysis. It allows you to drill down into data across multiple dimensions, returning the top N results at each level.

Example: Top Users by Region and Activity

Consider a `UserActivity` table with columns `Region`, `UserId`, and `ActivityType`. You want to find the top 2 regions and, within each, the top 3 users by activity count.

```
UserActivity
```

```
| where Timestamp > ago(30d)
```

```
| top-nested 2 of Region by RegionCount =  
count(),
```

```
    top-nested 3 of UserId by ActivityCount =  
count()
```

```
| project Region, RegionCount, UserId,  
ActivityCount
```

What it does: This query first finds the top 2 regions by activity count, then within each region, it finds the top 3 users by activity count. The result is a hierarchical view of the data.

Why you need it: `top-nested` is perfect for multi-level analysis, such as identifying top customers, regions, or products in a hierarchical structure, without writing multiple queries.

5. `evaluate bag_unpack`: Flattening Dynamic Columns

The `evaluate bag_unpack` operator (used with the `bag_unpack` plugin) is ideal for flattening dynamic columns (like JSON objects) into individual columns. This is particularly useful when dealing with semi-structured data.

Example: Unpacking JSON Metadata

Assume a `Events` table has a `Metadata` column containing JSON objects like `{"Environment": "Prod", "Version": "1.2.3"}`. You want to extract these fields as columns.

Events

```
| evaluate bag_unpack(Metadata)  
  
| summarize EventCount = count() by  
Environment, Version
```

What it does: The `bag_unpack` operator transforms the `Metadata` JSON into columns `Environment` and `Version`. The `summarize` operator then groups events by these new columns.

Why you need it: JSON data is common in modern datasets, but analyzing it can be cumbersome. `bag_unpack` makes it easy to convert JSON properties into queryable columns without manual parsing.

These five KQL operators—`mv-expand`, `make-series`, `parse`, `top-nested`, and `evaluate bag_unpack`—are powerful tools that can simplify complex data analysis tasks. Whether you're unraveling arrays, building time series, parsing strings, analyzing hierarchies, or flattening JSON, these operators

can save you time and effort. Next time you're writing a KQL query, consider whether one of these hidden gems can make your life easier.

Try them out in your next KQL query and unlock new insights from your data!

Chapter 23

Advanced Pattern Matching in Kusto Query Language (KQL): Unlocking Complex Log Parsing in Azure

Kusto Query Language (KQL), the powerhouse behind Azure Data Explorer, Azure Monitor, and Microsoft Sentinel, is a go-to for analyzing

massive datasets. One of its standout features is its robust string manipulation and pattern-matching capabilities, which shine when parsing complex logs. Whether you're troubleshooting application errors, hunting for security threats, or extracting insights from telemetry, mastering KQL's string functions like contains, matches regex, and parse can save hours of manual log sifting. In this post, we'll dive into these advanced pattern-matching techniques with practical Azure examples to supercharge your log analysis.

Why Pattern Matching Matters in KQL

Logs are often messy, unstructured, or semi-structured, packed with critical details buried in text. Pattern matching in KQL lets you extract, filter, and transform these details efficiently. For example, you might need to:

- Identify error codes in application logs.
- Extract IP addresses or user IDs from security events.
- Parse timestamps or URLs from telemetry data.

KQL's string functions make this possible with precision and flexibility. Let's explore three key

tools—contains, matches regex, and parse—and see them in action with real-world Azure scenarios.

1. **contains**: Quick and Simple String Searches

The contains operator is your first stop for straightforward pattern matching. It checks if a substring exists within a string, making it ideal for quick filters. It's case-insensitive by default, but you can toggle case sensitivity with `contains_cs`.

Example: Filtering Application Logs for Errors

Suppose you're analyzing logs in Azure Monitor's AppInsights table to find entries with the word "error". Here's a KQL query:

```
AppInsights
```

```
| where message contains "error"
```

```
| project timestamp, message
```

```
| limit 10
```

This query scans the message column for "error" and returns the timestamp and message for the top 10 matching logs. For case-sensitive searches, use:

AppInsights

```
| where message contains_cs "Error"
```

Pro Tip

Use !contains to exclude matches. For example, to ignore warnings but catch errors:

AppInsights

```
| where message contains "error" and message  
!contains "warning"
```

2. matches regex: Precision with Regular Expressions

When contains is too broad, matches regex brings surgical precision. Regular expressions (regex) let you define complex patterns, like specific formats for IPs, URLs, or error codes. KQL uses the .NET regex flavor, so you can leverage familiar syntax.

Example: Extracting IP Addresses from Security Logs

Imagine you're using Microsoft Sentinel to analyze security logs in the SecurityEvent table, and you need to extract IP addresses from event descriptions. An IPv4 address follows the pattern xxx.xxx.xxx.xxx. Here's a KQL query:

```
SecurityEvent
```

```
| where EventDescription matches regex  
@"\b\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3}\b"
```

```
| project EventID, EventDescription
```

This regex (`\b\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3}\b`) matches four groups of 1-3 digits separated by

dots, ensuring valid IP-like patterns. The \b ensures word boundaries to avoid false matches.

Bonus: Extracting Matches

To extract the IP itself, combine matches regex with extract:

```
SecurityEvent  
  
| extend IP =  
extract(@"\b(\d{1,3}.\d{1,3}.\d{1,3}.\d{1,3})\b", 1, EventDescription)  
  
| where isnotempty(IP)  
  
| project EventID, IP
```

Here, extract captures the first regex group (the IP) into a new column, IP.

3. parse: Structured Extraction from Semi-Structured Logs

The parse operator is a game-changer for semi-structured logs, letting you extract multiple fields in one go based on a pattern. It's perfect when logs follow a predictable format, like key-value pairs or delimited strings.

Example: Parsing IoT Telemetry Logs

Suppose you're working with IoT telemetry in Azure Data Explorer, stored in a table called IoTData. Each log entry has a message like: DeviceID: Sensor123, Temp: 25.5C, Timestamp: 2025-06-02T09:00:00Z. You want to extract DeviceID, Temp, and Timestamp as separate columns. Here's the KQL query:

```
IoTData
```

```
| parse message with "DeviceID: " DeviceID ",  
Temp: " Temp "C, Timestamp: " LogTimestamp  
  
| project DeviceID, Temp, LogTimestamp
```

This query:

- Matches the pattern in message.

- Extracts DeviceID, Temp, and LogTimestamp into new columns.
- Ignores literal text like "DeviceID: " and "C, Timestamp: ".

Handling Variations

If logs have inconsistent formats, use parse kind=regex for regex-based parsing:

```
IoTData
```

```
| parse kind=regex message with @"DeviceID:  
(\w+), Temp: ([\d.]+)C, Timestamp: (.+)"
```

```
| project DeviceID, Temp, LogTimestamp
```

This handles cases where DeviceID is alphanumeric, Temp is a decimal, and Timestamp varies.

Practical Azure Scenario: Combining Techniques

Let's tie it all together with a real-world example. You're a DevOps engineer analyzing Azure

Monitor logs to troubleshoot a web app. Logs in the AppInsights table contain messages like: Request failed for user: john.doe@company.com, URL: /api/v1/data, ErrorCode: 500.

Your goal: Extract the user email, URL, and error code, then filter for 500 errors. Here's the KQL query:

```
AppInsights
```

```
| parse message with "Request failed for user:  
" UserEmail ", URL: " URL ", ErrorCode: "  
ErrorCode  
  
| where ErrorCode == "500"  
  
| extend Domain = extract(@"(\\w+\\.\\w+)", 1,  
UserEmail)  
  
| project timestamp, UserEmail, Domain, URL,  
ErrorCode
```

This query:

- Uses parse to extract UserEmail, URL, and ErrorCode.
- Filters for ErrorCode == "500".
- Uses extract with regex to pull the email domain (e.g., company.com).
- Projects relevant columns for analysis.

Best Practices for KQL Pattern Matching

1. **Start Simple:** Use contains for quick filters before diving into regex or parse.
2. **Test Regex Incrementally:** Build and test regex patterns using tools like regex101.com, then adapt to KQL's syntax.
3. **Leverage extend for Clarity:** Create intermediate columns to break down complex parsing steps.
4. **Optimize for Performance:** Narrow your dataset with where clauses before applying parse or matches regex.
5. **Use Comments:** Add // comments in KQL to document complex queries for team collaboration.

KQL's pattern-matching capabilities—contains, matches regex, and parse—are indispensable for taming complex logs in Azure. Whether you're filtering errors with contains, extracting IPs with matches regex, or structuring IoT data with parse, these tools empower you to unlock insights fast. Try these examples in Azure Data Explorer or Azure Monitor, and experiment with your own logs to see the power of KQL in action.

Chapter 24

Extending KQL with Inline Functions: Simplifying Complex Queries

Kusto Query Language (KQL) is a powerful tool for querying and analyzing large datasets in Azure Data Explorer, Log Analytics, and other Azure services. As queries grow in complexity, maintaining readability and reusability becomes a challenge. Inline functions in KQL offer a solution by allowing you to encapsulate logic into

reusable components within your queries. In this chapter, we'll explore how to create inline functions in KQL, their benefits, and practical examples to simplify complex queries and improve maintainability.

What Are Inline Functions in KQL?

Inline functions in KQL are user-defined functions (UDFs) created within a query using the let statement. Unlike stored functions, which are saved in the database, inline functions are scoped to the query in which they are defined. They allow you to encapsulate repetitive or complex logic, making your queries more modular, readable, and easier to maintain.

Inline functions can take parameters, perform calculations, or transform data, and they can be invoked multiple times within the same query. This approach is particularly useful for simplifying complex logic, reducing code duplication, and improving query clarity.

Why Use Inline Functions?

Using inline functions in KQL provides several benefits:

1. **Improved Readability:** Break down complex queries into smaller, logical units that are easier to understand.
2. **Reusability:** Reuse the same logic multiple times within a query without rewriting code.
3. **Maintainability:** Centralize logic in one place, making it easier to update or debug.
4. **Modularity:** Encapsulate specific tasks, promoting a cleaner and more organized query structure.

Syntax for Creating Inline Functions

Inline functions are defined using the let statement in KQL. Here's the basic syntax:

```
let functionName = (param1: type1, param2: type2, ...) {
```

```
// Function logic here
```

```
// Return a result using expressions or queries
```

```
};
```

- functionName: The name of the function.
- param1, param2: Parameters with their respective data types (e.g., string, int, datetime).
- The function body contains the logic, which can include KQL expressions or queries.
- The function is invoked by calling functionName(arg1, arg2, ...).

Example 1: Simplifying String Manipulation

Let's start with a simple example. Suppose you frequently need to clean and format strings in your dataset, such as trimming whitespace and converting to uppercase. Instead of repeating the logic, you can create an inline function.

```
let CleanString = (input: string) {
```

```
    toupper(trim(input))
```

```
} ;  
  
// Example usage  
  
StormEvents  
  
| where StartTime > datetime(2007-01-01)  
  
| project EventType, CleanedEventType =  
CleanString(EventType)  
  
| limit 5
```

Explanation:

- The CleanString function takes a string parameter, applies trim() to remove whitespace, and toupper() to convert to uppercase.
- The function is invoked in the project operator to create a new column, CleanedEventType.

- This approach keeps the string manipulation logic in one place, making it reusable and easier to modify.

Example 2: Calculating Time Differences

For time-based analysis, you might need to calculate the duration between two datetime columns. An inline function can simplify this.

```
let CalculateDuration = (start: datetime, end: datetime) {  
    end - start  
};  
  
// Example usage  
  
StormEvents
```

```
| where StartTime > datetime(2007-01-01)

| project EventId, StartTime, EndTime, Duration
= CalculateDuration(StartTime, EndTime)

| limit 5
```

Explanation:

- The CalculateDuration function takes two datetime parameters and returns their difference.
- The function is used in the project operator to compute the Duration column.
- This eliminates repetitive datetime subtraction logic and improves query clarity.

Example 3: Complex Filtering Logic

For more complex scenarios, inline functions can encapsulate filtering logic. Suppose you want to categorize storm events based on damage costs into "Low," "Medium," or "High" severity levels.

```
let CategorizeDamage = (damageCost: long) {  
  
    iff(damageCost < 10000, "Low",  
  
        iff(damageCost < 100000, "Medium",  
            "High"))  
  
};
```

```
// Example usage
```

```
StormEvents
```

```
| where StartTime > datetime(2007-01-01)
```

```
| project EventId, DamageProperty, Severity =  
CategorizeDamage(DamageProperty)
```

```
| limit 5
```

Explanation:

- The CategorizeDamage function takes a long parameter (damageCost) and uses nested iff() statements to categorize the cost.
- The function is applied in the project operator to create a Severity column.
- This approach centralizes the categorization logic, making it easy to adjust thresholds or add new categories.

Example 4: Reusing Query Logic

Inline functions can also encapsulate entire query fragments. Suppose you need to filter and summarize storm events by state multiple times in a query. You can define a function to handle this.

```
let SummarizeByState = (stateName: string) {  
    StormEvents  
    | where StartTime > datetime(2007-01-01)  
    | groupby StateName  
    | sum DamageCost  
    | order by sum(DamageCost) desc  
    | take 10  
}
```

```
| where State == stateName  
  
| summarize TotalEvents = count() ,  
AvgDamage = avg(DamageProperty) by State  
  
};  
  
// Example usage  
  
union  
  
(SummarizeByState("FLORIDA")) ,  
  
(SummarizeByState("TEXAS")) ,  
  
(SummarizeByState("CALIFORNIA"))
```

Explanation:

- The SummarizeByState function takes a string parameter (stateName) and returns a summarized result for the specified state.
- The union operator combines results from multiple function calls.
- This approach avoids duplicating the summarization logic for each state, improving maintainability.

Best Practices for Inline Functions

1. **Keep Functions Focused:** Each function should have a single, clear purpose to maximize reusability.
2. **Use Descriptive Names:** Choose meaningful function names that reflect their purpose (e.g., CleanString, CategorizeDamage).
3. **Validate Parameters:** Ensure parameters have appropriate types and handle edge cases (e.g., null values) if necessary.
4. **Test Functions Independently:** Before integrating a function into a larger query, test it with sample data to verify correctness.
5. **Consider Stored Functions for Reusability:** If a function will be used across multiple queries, consider

defining it as a stored function in the database instead of an inline function.

Limitations of Inline Functions

While inline functions are powerful, they have some limitations:

- **Query-SScoped**: Inline functions are only available within the query where they are defined. For broader reuse, use stored functions.
- **Performance**: Complex functions may impact query performance, especially if they process large datasets. Test and optimize as needed.
- **No Recursion**: KQL does not support recursive function calls.

Inline functions in KQL are a game-changer for simplifying complex queries and improving maintainability. By encapsulating repetitive or intricate logic into reusable components, you can make your queries more readable, modular, and easier to update. Whether you're cleaning data, calculating metrics, or applying complex filtering, inline functions help you write cleaner and more efficient KQL code.

Start experimenting with inline functions in your next KQL query, and see how they can streamline your data analysis workflows. If you have examples of how you've used inline functions in KQL, please share them!

Happy querying!

Chapter 25

Pattern Matching with KQL Regular Expressions: Using the Parse and Extract Functions to Clean Data

Pattern matching is a vital aspect of data analysis, empowering users to identify and isolate specific elements within textual and numerical datasets. In the realm of KQL (Kusto Query Language), regular expressions provide sophisticated methods for cleaning and transforming data. Two fundamental functions, parse and extract, offer powerful utilities for leveraging regular expressions in KQL to achieve precision and efficiency in data manipulation.

Let's delve into the mechanics of these functions, showcasing their application through working examples. By the end, you will have a robust understanding of how to use KQL's regular expressions to clean and process raw data effectively.

Overview of Regular Expressions in KQL

Regular expressions are sequences of characters that define search patterns. They are widely used in programming languages for text parsing, pattern recognition, and data extraction. In KQL, regular expressions are incorporated through specific functions that facilitate querying and transforming large datasets.

The major advantage of KQL is its simplicity and optimized performance for handling queries in Azure Data Explorer and other services.

Understanding how to apply regular expressions effectively can unlock new possibilities for transforming unstructured or semi-structured data.

The parse Function

The parse function in KQL is primarily used to break down a string into structured components based on a specific pattern. It requires the definition of a pattern that includes named fields to extract information.

Syntax

The syntax for the parse function is as follows:

```
parse [kind=regex] ColumnName with Pattern
```

Here:

- *ColumnName* refers to the source column containing the data to parse.
- *Pattern* is the regular expression defining the structure of the data.

Example: Parsing a Log Entry

Consider a dataset containing server log entries in the following format:

```
2025-05-20 12:00:00 INFO UserID=1234  
Action=Login
```

To extract the date, time, user ID, and action, you can use the parse function:

```
datatable(Log: string)
```

```
[
```

```
"2025-05-20 12:00:00 INFO UserID=1234  
Action=Login",
```

```
"2025-05-20 12:30:00 ERROR UserID=5678  
Action=Logout"
```

```
]
```

```
| parse Log with Date:string " " Time:string "  
" Level:string " UserID=" UserID:int " Action="  
Action:string
```

This results in:

```
Date | Time | Level | UserID | Action
```

```
-----  
----
```

2025-05-20 | 12:00:00 | INFO | 1234 | Login

2025-05-20 | 12:30:00 | ERROR | 5678 | Logout

The extract Function

The extract function in KQL is designed to retrieve specific segments of text matching a given pattern. Unlike parse, extract does not require named fields and is ideal for isolating individual components within a dataset.

Syntax

The syntax for the extract function is:

```
extract(regex, captureGroup, ColumnName)
```

Here:

- *regex* is the regular expression defining the pattern.
- *captureGroup* is the group within the regex to extract.
- *ColumnName* represents the column containing the target data.

Example: Extracting Email Addresses

Suppose you have a dataset containing user messages with embedded email addresses:

```
Message: "Contact support at help@company.com  
for assistance."
```

To extract the email address, use the `extract` function:

```
datatable(Message: string)
```

```
[
```

```
"Contact support at help@company.com for  
assistance.",
```

```
"Reach out to admin@domain.org for further  
help."
```

```
]
```

```
| extend Email = extract(@"(\w+\@\w+\.\w+)", 1,  
Message)
```

The output will be:

Message	Email
----- -----	
Contact support at help@company.com for assistance.	help@company.com
Reach out to admin@domain.org for further help.	admin@domain.org

Applications in Data Cleaning

The parse and extract functions can be utilized for various data cleaning tasks, such as:

- Removing unwanted characters and formatting inconsistencies

- Standardizing data fields based on patterns
- Isolating specific elements for further analysis

Example: Cleaning Phone Numbers

Imagine a dataset containing phone numbers in inconsistent formats:

Numbers: "+1-202-555-0173", "202 555 0198",
"(202) 555-0147"

To standardize these phone numbers, you can use the extract function:

```
datatable(PhoneNumber: string)
```

```
[
```

```
+1-202-555-0173,
```

```
"202 555 0198",
```

```
"(202) 555-0147"
```

```
]
```

```
| extend CleanedNumber = extract(@"(\d{3}) [-] ?(\d{3}) [-] ?(\d{4})", 0, PhoneNumber)
```

This results in:

```
PhoneNumber | CleanedNumber
```

```
----- | -----
```

```
+1-202-555-0173 | 2025550173
```

```
202 555 0198 | 2025550198
```

```
(202) 555-0147 | 2025550147
```

Key Considerations

While using regular expressions with KQL functions, keep the following in mind:

- Patterns should be precise to avoid errors in data extraction.
- The parse function is ideal for structured data with consistent elements.
- The extract function is better suited for datasets requiring selective extraction.
- Test your regular expressions thoroughly to ensure accurate results.

Mastering pattern matching with regular expressions in KQL, particularly through the parse and extract functions, is an essential skill for data professionals working with Azure Data Explorer and similar platforms. These tools simplify the process of converting raw, unstructured data into meaningful, clean datasets ready for analysis.

By experimenting with the examples provided in this chapter, you can develop a deeper understanding of how these functions work and how they can be tailored to your specific data transformation needs. With practice, you'll be

able to leverage KQL to unlock valuable insights from complex datasets.

Chapter 26

Exploring the Power of the KQL Evaluate Operator

The **Kusto Query Language (KQL)** is a powerful tool for querying large datasets in Azure Data Explorer and Microsoft Fabric. Among its many operators, the **Evaluate operator** stands out for its ability to invoke advanced query extensions, known as plugins. This operator is essential for performing complex functions, including machine learning models, which require a deeper understanding of KQL extensions.

What is the Evaluate Operator?

The **Evaluate operator** is a tabular operator that allows users to invoke service-side query extensions. These extensions, or plugins, are not bound by the relational nature of KQL, meaning they can produce dynamic output schemas based on the data they process.

Syntax and Parameters

The syntax for the Evaluate operator is straightforward:

```
[T | ] evaluate [evaluateParameters] PluginName  
([PluginArgs])
```

- **T**: A tabular input to the plugin.
- **evaluateParameters**: Space-separated parameters controlling the behavior of the operation.
- **PluginName**: The name of the plugin being invoked.
- **PluginArgs**: Arguments provided to the plugin.

Key Plugins

The Evaluate operator supports various plugins, each designed for specific tasks. Some notable plugins include:

- **autocluster**: Automatically clusters data based on similarities.
- **pivot**: Transforms data from rows to columns.
- **R plugin**: Integrates R scripts for advanced statistical analysis.
- **rolling-percentile**: Calculates rolling percentiles over a dataset.

Distribution Hints

Distribution hints specify how the plugin execution is distributed across cluster nodes:

- **single**: A single instance runs over the entire query data.
- **per_node**: Instances run on each node over the data they contain.
- **per_shard**: Instances run over each shard of the data.

NOTE: To use the following queries,
see: [Sample Gallery](#)

Example 1: Using the `autocluster` Plugin

This example demonstrates how to automatically cluster data based on similarities using the `autocluster` plugin.

StormEvents

```
| evaluate autocluster()
```

This query takes the `StormEvents` table and applies the `autocluster` plugin to group similar events together.

Example 2: Using the `pivot` Plugin

This example shows how to transform data from rows to columns using the `pivot` plugin.

```
StormEvents
```

```
| summarize Count = count() by State, EventType  
  
| evaluate pivot(State, EventType, Count)
```

Here, the `pivot` plugin is used to create a pivot table that shows the count of events by state and event type.

Example 3: Using the `R` Plugin for Advanced Statistical Analysis

This example integrates an R script to perform advanced statistical analysis.

```
StormEvents
```

```
| evaluate script("R",
"summary(lm(DamageProperty ~ State + EventType,
data))")
```

In this query, the `R` plugin runs an R script that performs a linear regression analysis on the `DamageProperty` based on `State` and `EventType`.

Example 4: Using the `rolling-percentile` Plugin

This example calculates rolling percentiles over a dataset.

```
StormEvents
```

```
| evaluate rolling_percentile(90, Timestamp,
DamageProperty)
```

This query calculates the 90th percentile of `DamageProperty` values over time.

Example 5: Using Distribution Hints

This example demonstrates how to use distribution hints with the `pivot` plugin.

```
StormEvents
```

```
| summarize Count = count() by State, EventType  
  
| evaluate hint.distribution = per_node  
pivot(State, EventType, Count)
```

In this query, the `hint.distribution = per_node` hint is used to distribute the `pivot` operation across nodes.

The **Evaluate operator** is a versatile tool in KQL, enabling advanced functions and machine learning models. By understanding its syntax, parameters, and supported plugins, users can leverage its full potential to perform complex data analysis tasks.

Chapter 27

Visualizing with KQL: Best Practices for Render and Timechart Operators

Kusto Query Language (KQL) is a powerful tool for querying and analyzing large datasets, particularly in Azure Data Explorer (ADX) and Microsoft Sentinel. When it comes to transforming complex datasets into intuitive dashboards, the render and timechart operators are your go-to tools for creating meaningful visualizations. In this chapter, we'll explore best practices for using these operators effectively, with practical examples to help you craft clear, actionable dashboards.

Why Visualize with KQL?

Visualizations turn raw data into insights. Dashboards built with KQL allow analysts to spot trends, identify anomalies, and communicate findings efficiently. The render operator in KQL specifies how data should be visualized (e.g., as charts, tables, or graphs), while timechart is

tailored for time-series visualizations, making it ideal for monitoring trends over time. Together, they enable you to transform complex datasets into intuitive, interactive dashboards.

Best Practices for Using `render` and `timechart`

1. Understand Your Data and Audience

Before writing a KQL query, clarify the purpose of your visualization and who will use it. Are you tracking system performance for engineers or presenting high-level trends to executives? This informs your choice of chart type and level of detail.

- **Tip:** Use `render` to select chart types that match your data's story. For example, use `linechart` for trends, `barchart` for comparisons, or `piechart` for proportional data.
- **Example:** Suppose you're analyzing login attempts to detect potential security issues. A `timechart` with a `linechart` `render` can highlight spikes in failed logins over time, making it easier to spot anomalies.

```
SecurityEvent
```

```
| where EventID == 4625  
  
| summarize FailedLogins = count() by  
bin(TimeGenerated, 1h)  
  
| render timechart
```

This query counts failed login attempts per hour and visualizes them as a line chart, ideal for security analysts monitoring threats.

2. Leverage timechart for Time-Series Data

The timechart operator is designed for time-series analysis, automatically binning data by time intervals. Use it when your dataset includes a timestamp column (e.g., TimeGenerated in ADX).

- **Best Practice:** Always use the bin() function to define appropriate time intervals (e.g., 1h, 1d) to avoid overly granular or cluttered charts.

- **Example:** To monitor CPU usage across servers over a week, aggregate data by day:

Perf

```
| where CounterName == "% Processor Time"  
  
| summarize AvgCPU = avg(CounterValue) by  
bin(TimeGenerated, 1d), Computer  
  
| render timechart
```

This creates a line chart showing average CPU usage per server, with each line representing a server. The daily binning smooths out noise, making trends clearer.

3. Simplify with Aggregation

Complex datasets often contain too much raw data for effective visualization. Use summarize to aggregate data before rendering, reducing noise and focusing on key metrics.

- **Tip:** Combine summarize with functions like count(), avg(), sum(), or max() to extract meaningful metrics.
- **Example:** To visualize the top 5 event types in a log dataset:

```
SecurityEvent
```

```
| summarize EventCount = count() by EventID  
| top 5 by EventCount  
  
| render piechart
```

This query counts occurrences of each EventID, selects the top 5, and displays them in a pie chart, making it easy to see which events dominate the dataset.

4. Choose the Right Chart Type

The render operator supports various chart types, including table, barchart, columnchart, piechart, areachart, linechart, scatterchart, and timechart.

Select the chart that best conveys your data's message.

- **Best Practice:** Avoid overloading dashboards with too many chart types. Stick to 2-3 types for consistency and clarity.
- **Example:** To compare total requests by region:

WebLogs

```
| summarize TotalRequests = count() by Region  
  
| render barchart
```

A bar chart is ideal here, as it clearly compares discrete categories (regions) side by side.

5. Use Annotations and Titles

Clear labels and titles make dashboards intuitive. Use the `with` clause in `render` to customize chart properties like titles, axis labels, and legends.

- **Tip:** Always include a descriptive title and label axes to avoid ambiguity.
- **Example:** To visualize network traffic with clear annotations:

```
NetworkTraffic
```

```
| summarize TotalBytes = sum(BytesTransferred)  
by bin(TimeGenerated, 1h)  
  
| render timechart with (title="Network Traffic  
Over Time", ytitle="Bytes Transferred",  
xtitle="Time")
```

This creates a time chart with a title and labeled axes, ensuring viewers understand the data at a glance.

6. Handle Missing Data Gracefully

Time-series data often has gaps, which can distort visualizations. Use make-series to fill in missing time bins with default values (e.g., 0) for smoother charts.

- **Example:** To visualize application errors with filled gaps:

```
AppErrors
```

```
| make-series ErrorCount = count() default 0 on  
TimeGenerated from ago(7d) to now() step 1h by  
AppName
```

```
| render timechart
```

This ensures a continuous line chart, even if some time bins have no errors, making trends easier to interpret.

7. Optimize for Performance

Large datasets can slow down queries and dashboards. Optimize by filtering early, limiting time ranges, and sampling when appropriate.

- **Tip:** Use where clauses to narrow down data before aggregation, and consider take or sample for quick prototyping.
- **Example:** To analyze recent web requests efficiently:

```
WebLogs
```

```
| where TimeGenerated > ago(1d)

| summarize RequestCount = count() by
bin(TimeGenerated, 10m)

| render timechart
```

This query filters to the last 24 hours, reducing the dataset size while still providing a detailed view of request trends.

8. Test and Iterate

Dashboards evolve with user feedback. Test your visualizations with stakeholders to ensure they meet their needs, and refine queries as new requirements emerge.

- **Tip:** Use ADX's dashboard feature to pin multiple queries and create interactive views. Adjust chart types or time ranges based on user input.

Example: Building an Intuitive Dashboard

Let's combine these practices to create a dashboard for monitoring a web application. The dashboard will include:

1. A time chart of request latency.
2. A pie chart of error types.
3. A bar chart of top 5 active users.

```
// Time chart: Average request latency
```

```
WebLogs
```

```
| where TimeGenerated > ago(1d)
```

```
| summarize AvgLatency = avg(RequestDurationMs)  
by bin(TimeGenerated, 10m)
```

```
| render timechart with (title="Average Request  
Latency", ytitle="Latency (ms)", xtitle="Time")
```

```
// Pie chart: Distribution of error types

WebLogs

| where ResponseCode >= 400

| summarize ErrorCount = count() by
ResponseCode

| render piechart with (title="Error Types by
Response Code")

// Bar chart: Top 5 active users

WebLogs

| summarize RequestCount = count() by UserId
```

```
| top 5 by RequestCount
```

```
| render barchart with (title="Top 5 Active  
Users", ytitle="Requests", xtitle="User ID")
```

This dashboard provides a holistic view: latency trends for performance monitoring, error distribution for debugging, and user activity for engagement insights. Each visualization is clear, labeled, and optimized for quick loading.

The render and timechart operators in KQL are powerful tools for transforming complex datasets into intuitive dashboards. By understanding your data, choosing appropriate chart types, aggregating effectively, and optimizing performance, you can create visualizations that drive insights and action. Start with small queries, test with your audience, and iterate to build dashboards that tell a compelling story.

For more KQL tips, check out the [Azure Data Explorer documentation](#) or share your favorite visualization techniques on [X](#)!

Chapter 28

Advanced Joins in KQL: Going Beyond the Inner Loop

Kusto Query Language (KQL), the powerhouse behind Azure Data Explorer, offers a robust set of tools for querying large datasets. While many users start with basic inner joins to combine tables, KQL's join capabilities go far deeper, providing nuanced options for handling matches, non-matches, and performance optimizations. In this post, we'll dive into the different join types, explore anti-joins for exclusionary logic, and cover lookup tables for efficient dimension enrichment. We'll break down the nuances of each, highlight differences from traditional SQL, and include practical examples to illustrate their use.

Whether you're analyzing logs, telemetry, or time-series data, mastering these advanced joins can transform your queries from simple filters to sophisticated data pipelines. Let's get started.

Understanding the Core Join Types in KQL

KQL's join operator merges two datasets based on specified conditions, but the "kind" parameter dictates how rows and schemas are handled.

Unlike SQL, where INNER JOIN is the default, KQL defaults to *innerunique*, which introduces deduplication on the left side. This can be a gotcha for SQL veterans, as it prevents unexpected row explosions from duplicates.

Here's a quick overview of the primary join kinds:

- **innerunique (default)**: Performs an inner join but deduplicates the left table's join keys before matching. This reduces output size when the left side has repeats but you only care about unique combinations.
- **inner**: A standard inner join, returning only rows where there's a match in both tables, without any deduplication.
- **leftouter**: Returns all rows from the left table, with matching rows from the right; non-matches get null values on the right side.
- **rightouter**: The mirror of leftouter, preserving all rows from the right table.

- **fullouter**: Combines both sides fully, filling non-matches with nulls—ideal for complete comparisons.

Nuances to note:

- Schema merging includes all columns from both tables, with join keys appearing once (unless aliased).
- Performance considerations: Joins can be resource-intensive on large datasets. Use hints like
`hint.shufflekey=KeyColumn` for distributed processing when join keys have high cardinality. KQL assumes the right table is smaller by default; for the opposite, use `hint.strategy=broadcast` on the left.
- Differences from SQL: No automatic deduplication in SQL's INNER JOIN, and KQL lacks CROSS JOIN (use `join kind=inner` without conditions for similar behavior, but cautiously).

Example Setup

For our examples, we'll use two simple in-memory tables:

```
let Employees = datatable(EmployeeID: int,  
Name: string, DeptID: int) [
```

```
    1, "Alice", 10,
```

```
    2, "Bob", 20,
```

```
    3, "Charlie", 10,
```

```
    4, "David", 30,
```

```
    5, "Eve", 10 // Duplicate DeptID for demo
```

```
] ;
```

```
let Departments = datatable(DeptID: int,  
DeptName: string) [
```

```
    10, "HR",
```

```
20, "Engineering",  
  
40, "Marketing" // No match for DeptID 30  
  
];
```

Innerunique Join

Employees

```
| join kind=innerunique Departments on DeptID
```

Output (deduplicates left on DeptID, so only unique DeptIDs from Employees match):

This is great for avoiding bloat in results but might surprise you if duplicates are intentional.

Inner Join

Employees

```
| join kind=inner Departments on DeptID
```

Output (all matches, including duplicates):

EmployeeID	Name	DeptID	DeptID1	DeptName
1	Alice	10	10	HR
3	Charlie	10	10	HR
5	Eve	10	10	HR
2	Bob	20	20	Engineering

No deduplication—every match is included.

Leftouter Join

Employees

```
| join kind=leftouter Departments on DeptID
```

Output (all Employees, nulls for non-matches like David):

EmployeeID	Name	DeptID	DeptID1	DeptName
1	Alice	10	10	HR
3	Charlie	10	10	HR
5	Eve	10	10	HR
2	Bob	20	20	Engineering
4	David	30		

Rightouter and fullouter follow similar patterns but flip or combine preservation.

These examples are inspired by Microsoft documentation on the join operator.

Semi-Joins: Filtering Without the Extra Columns

Semi-joins are like inner joins but only return columns from one side, acting as efficient filters. They're perfect when you want to subset a table based on existence in another without bloating your schema.

- **leftsemi**: Returns left table rows that have matches in the right, but only left columns.
- **rightsemi**: The opposite, returning right table rows with matches, only right columns.

Nuances:

- No schema merge—keeps things lean.
- Similar to SQL's EXISTS subquery, but more performant in distributed systems like KQL.
- Use when you need filtering logic without the overhead of full joins.

Leftsemi Example

Employees

```
| join kind=leftsemi Departments on DeptID
```

Output (Employees with existing departments, left columns only):

EmployeeID	Name	DeptID
1	Alice	10
2	Bob	20
3	Charlie	10
5	Eve	10

David is excluded since DeptID 30 has no match.

Rightsemi would filter Departments to only those with Employees (e.g., HR and Engineering).

Anti-Joins: The Power of Exclusion

Anti-joins are your go-to for "not in" scenarios, returning rows from one side that *don't* match the other. They're invaluable for data cleaning, anomaly detection, or diffing datasets.

- **leftanti (or anti, leftantisemi)**: Returns left rows with no right matches, left columns only.
- **rightanti (or rightantisemi)**: Returns right rows with no left matches, right columns only.

Nuances:

- Essentially the inverse of semi-joins.

- Differs from SQL's NOT EXISTS by being a direct operator, often faster in KQL's columnar storage.
- Tip: For large datasets, combine with summarize or where to handle nulls post-join. Avoid if either side is massive without shuffle hints, as mismatches can still scan everything.

Leftanti Example

Employees

```
| join kind=leftanti Departments on DeptID
```

Output (Employees without departments):

EmployeeID	Name	DeptID
4	David	30

Rightanti Example

Employees

```
| join kind=rightanti Departments on DeptID
```

Output (Departments without employees):

DeptID	DeptName
40	Marketing

These are based on behaviors detailed in KQL's join documentation. Use anti-joins sparingly on huge tables, as they don't short-circuit like some SQL engines.

Lookup Tables: Optimized for Dimension Enrichment

When you have a large fact table and a small dimension table (e.g., lookup codes or categories), the lookup operator shines. It's similar to a leftouter or inner join but optimized for broadcasting the small table across nodes.

Syntax: LeftTable | lookup
kind=(leftouter|inner) (RightTable) on
Attributes

Nuances:

- Assumes left is large (facts), right is small (dimensions)—opposite of join's default.
- Doesn't repeat join keys in output; auto-broadcasts right table.
- Limited to leftouter (default) or inner; fails if right exceeds ~tens of MB (check with summarize sum(estimate_data_size(*))).
- Differences from join: Leaner output, better perf for small lookups, but no other kinds supported.
- When to use: Enrich logs with user metadata or error codes without full join overhead.

Lookup Example (from docs)

```
let FactTable = datatable(Row: string,  
Personal: string, Family: string) [  
  
    "1", "Rowan", "Murphy",  
  
    "2", "Ellis", "Turner",
```

```
"3", "Ellis", "Turner",
```

```
"4", "Maya", "Robinson",
```

```
"5", "Quinn", "Campbell"
```

```
] ;
```

```
let DimTable = datatable(Personal: string,  
Family: string, Alias: string) [
```

```
"Rowan", "Murphy", "rowanm",
```

```
"Ellis", "Turner", "ellist",
```

```
"Maya", "Robinson", "mayar",
```

```
"Quinn", "Campbell", "quinnnc"
```

```
];
```

```
FactTable | lookup kind=leftouter DimTable on  
Personal, Family
```

Output:

Row	Personal	Family	Alias
1	Rowan	Murphy	rowanm
2	Ellis	Turner	elist
3	Ellis	Turner	elist
4	Maya	Robinson	mayar
5	Quinn	Campbell	quinnnc

This efficiently adds the Alias without duplicating keys.

Choosing the Right Tool

Advanced joins in KQL empower you to handle complex data relationships with precision. Stick to inner or innerunique for matches, outer joins for preservation, semi/anti for filtering, and lookup for quick enrichments. Always consider

data sizes—profile with `estimate_data_size` and use hints for scale.

Experiment in your Azure Data Explorer cluster, and remember: the best join is the one that fits your data's story. For more details, check the official KQL docs on joins and lookups. Happy querying!

Chapter 29

Leveraging KQL to Analyze Malware Trends and Identify Recurring Threats

Gaining actionable insights from data is crucial for fortifying defenses. As organizations amass vast amounts of security logs and telemetry, the ability to extract meaningful patterns and trends from this data can be a game-changer. Enter KQL—Kusto Query Language—a powerful tool that empowers cybersecurity professionals to analyze, visualize, and act upon data with precision. This chapter delves into how KQL can help analyze trends in malware detections over time and identify repeat offenders or recurring attack vectors, enabling a more strategic approach to defense.

Analyzing Trends in Malware Detections Over Time

Understanding the ebb and flow of malware detections is essential for adjusting your security posture. By analyzing trends, you can uncover

seasonal patterns, the emergence of new threats, or the persistence of certain malware families. Here's how KQL can help:

NOTE: These queries are examples only. There is no MalwareDetections table.

Step 1: Aggregating Malware Detection Data

The first step in analyzing malware trends is to aggregate and visualize detection data over a specific period. For example, a KQL query might look like this:

```
MalwareDetections
```

```
| where Timestamp between (datetime(2025-01-01)
.. datetime(2025-04-01))

| summarize DetectedCount = count() by
bin(Timestamp, 1d), MalwareType

| render timechart
```

This query:

- Filters the `MalwareDetections` table for a specified time range.
- Groups detections by day (`bin(Timestamp, 1d)`) and malware type.
- Visualizes the data as a time chart, highlighting detection spikes and trends.

Step 2: Identifying Anomalies

Spikes in detection rates often indicate active campaigns or the introduction of a new malware variant. To further refine your insights, use KQL to look for anomalies:

```
MalwareDetections
```

```
| summarize DailyCount = count() by  
bin(Timestamp, 1d)
```

```
| extend AvgCount = avg(DailyCount)
```

```
| where DailyCount > 2 * AvgCount
```

This query pinpoints dates where malware detections significantly exceed the average, signaling potential incidents requiring immediate attention.

Identifying Repeat Offenders or Recurring Attack Vectors

Beyond trends, identifying persistent attackers or recurring vectors is key to preempting future threats. KQL enables you to drill down into the data for actionable insights.

Step 1: Tracking Repeat Offenders

A common tactic in threat hunting is to identify IP addresses or entities repeatedly involved in malicious activities. The following query uncovers repeat offenders:

```
MalwareDetections
```

```
| summarize OffenseCount = count() by  
AttackerIP
```

```
| where OffenseCount > 5  
  
| sort by OffenseCount desc
```

This query:

- Groups detections by the attacking IP ('AttackerIP').
- Filters for entities involved in more than 5 detections.
- Provides a ranked list of repeat offenders.

Step 2: Investigating Recurring Attack Vectors

Understanding how attacks are delivered—whether through phishing, malicious URLs, or exploit kits—can shape your defensive strategies. Use KQL to identify recurring vectors:

MalwareDetections

```
| summarize VectorCount = count() by DeliveryVector
```

```
| where VectorCount > 10  
  
| sort by VectorCount desc
```

This query highlights the most frequently used methods of malware delivery, enabling you to prioritize defenses against the most common threats.

Strategic Implications

KQL arms cybersecurity teams with the ability to transform raw data into strategic insights. By analyzing malware detection trends, you can:

- Allocate resources to peak threat periods.
- Adjust signature updates and rule sets to counter emerging threats.
- Bolster defenses against the most prevalent delivery methods.

Similarly, identifying recurring offenders and vectors allows you to:

- Block or monitor high-risk IPs and domains.
- Fine-tune access controls and email filters.

- Develop targeted training programs to mitigate recurring attack patterns.

In the dynamic landscape of cybersecurity, data-driven strategies are your best defense. KQL offers a robust framework to analyze trends, uncover patterns, and make informed decisions. Whether you're tracking malware detections over time or pinpointing persistent adversaries, KQL empowers you to stay one step ahead. By integrating these insights into your strategy, you can not only respond to threats but anticipate and prevent them, securing your organization's future.

Chapter 30

Anomaly Detection with KQL: Leveraging Time-Series Analysis for Security Insights

Anomaly detection plays a critical role in identifying threats and mitigating risks in real-time. Time-series analysis provides a powerful framework for detecting irregular patterns in data, particularly when dealing with continuous streams, such as logs, network traffic, or application telemetry. In this chapter, we explore how KQL (Kusto Query Language) can be leveraged to perform anomaly detection via time-series analysis, delivering actionable security insights.

Understanding KQL

KQL, or Kusto Query Language, is the query language used in Azure Data Explorer and Microsoft Sentinel, among other platforms. It is designed for querying large datasets efficiently and supports aggregations, filtering, and advanced analytics. With its intuitive syntax and

robust capabilities, KQL allows security analysts to query and analyze logs at scale.

Anomaly Detection and Time-Series Analysis

Time-series analysis focuses on data points indexed in time order. Common use cases include detecting spikes in network traffic, identifying drops in system performance, or catching unusual login attempts. By applying KQL to time-series data, analysts can uncover patterns that deviate from normal behavior, signaling potential security incidents.

Key Techniques in KQL for Time-Series Anomaly Detection

1. Aggregations

Aggregations such as *summarize* enable you to compute metrics over specific time intervals, such as averages, counts, or sums. This helps in

understanding baseline behavior and detecting deviations.

2. Time Binning

The `bin()` function is used to divide time into intervals (buckets), allowing for clear visualization of data trends. This is vital for tracking metrics over time.

3. Filters and Thresholds

KQL allows filtering data using logical operators. You can define thresholds for normal behavior and isolate data points that exceed these bounds.

4. Machine Learning Models

Microsoft Sentinel supports integration with AI models for anomaly detection, but simple methods like moving averages or standard deviation calculations can be implemented directly within KQL to identify anomalies.

Working Examples

Below are practical examples that demonstrate how KQL can be used for anomaly detection.

Example 1: Detecting Unusual Login Attempts

This query monitors login activities and identifies anomalies based on login frequency spikes.

```
SigninLogs
```

```
| summarize LoginCount = count() by  
bin(TimeGenerated, 1h)
```

```
| where LoginCount > 100
```

```
| order by TimeGenerated asc
```

Here, we:

- Summarize login counts using hourly bins.

- Flag anomalies where login counts exceed 100 within an hour.

Example 2: Identifying Data Transfer Spikes

This query tracks network data transfer volumes, alerting on significant spikes.

NetworkLogs

```
| summarize DataTransferred = sum(Bytes) by  
bin(TimeGenerated, 5m)  
  
| extend Baseline = avg(DataTransferred)  
  
| where DataTransferred > 2 * Baseline  
  
| order by TimeGenerated desc
```

Here, we:

- Summarize data transferred every 5 minutes.
- Calculate a baseline average for comparison.
- Flag intervals where data transfer exceeds twice the baseline.

Example 3: Monitoring Failed Authentication Attempts

This query monitors failed authentication attempts and highlights time intervals with sharp increases.

```
SigninLogs
```

```
| where ResultType == "Failure"
```

```
| summarize Failures = count() by  
bin(TimeGenerated, 10m)
```

```
| extend Baseline = avg(Failures)
```

```
| where Failures > 3 * Baseline  
  
| order by TimeGenerated asc
```

Here, we:

- Filter logs for failed authentication events.
- Count failures in 10-minute bins.
- Detect intervals with failure counts significantly above the baseline.

Visualization and Insights

KQL queries can be paired with visualization tools like Azure Monitor or Microsoft Sentinel to render graphs. Visualizing anomalies in time-series data provides intuitive insights into patterns, enabling faster responses to threats.

Best Practices

- **Define Clear Baselines:** Understand normal behavior before setting thresholds for anomalies.

- **Use Appropriate Aggregations:** Choose time intervals that align with your data's frequency and variability.
- **Combine with Automation:** Integrate KQL queries with workflows to trigger alerts or automated responses.
- **Test and Refine:** Continuously refine queries to enhance detection accuracy and reduce false positives.

Anomaly detection through KQL and time-series analysis empowers security teams with precise tools to identify and respond to irregularities in data. Whether detecting unusual login patterns or monitoring network spikes, leveraging KQL can enhance your security posture. By adopting the techniques and examples outlined in this chapter, organizations can build robust defenses against emerging threats while optimizing their operations.

Start applying time-series analysis with KQL today and unlock actionable security insights from your data streams!

Chapter 31

Handling the Entities column in the SecurityAlert table as a Top-Level Field

The **Entities** column in the **SecurityAlert** table is typically stored as a JSON array, meaning each entity is nested within the structure. To make each entity accessible as either a top-level field or a key-value pair under a single column, you can use **mv-expand** and **parse_json** to break down the array and structure it properly.

Approach 1: Expanding Entities into Separate Rows

If you want each entity to be a top-level field, you can expand the array into multiple rows:

```
SecurityAlert
```

```
| extend EntitiesDynamicArray =  
| parse_json(Entities)
```

```
| mv-expand EntitiesDynamicArray  
  
| extend EntityType =  
  tostring(EntitiesDynamicArray.Type),  
  
    EntityValue =  
  tostring(EntitiesDynamicArray)
```

This will allow you to work with each entity separately.

Approach 2: Structuring as Key-Value Pairs

If you prefer to keep all entities under a single column but accessible by key, you can use **bag_unpack()**:

```
SecurityAlert
```

```
| extend EntitiesDynamicObject =  
  parse_json(Entities)  
  
| evaluate bag_unpack(EntitiesDynamicObject)
```

This will convert the JSON properties into individual columns, making them accessible by key.

Chapter 32

Using KQL to Track System Vulnerabilities

System vulnerabilities are a persistent concern in the evolving landscape of cybersecurity. Outdated software, misconfigured systems, and improper administrative access are prime targets for exploitation by malicious actors. Enter KQL, or Kusto Query Language, a powerful tool for data analysis and security monitoring within platforms like Azure Monitor and Microsoft Sentinel. This chapter explores how KQL can be used to identify system vulnerabilities, detect privilege escalation attempts, and enhance your overall security posture.

Tracking Outdated Software and Configurations

One of the most common vulnerabilities in a system is outdated software or configurations. These weaknesses often serve as entry points for

attackers, making it crucial to identify and address them promptly.

Why Outdated Software Matters

Outdated software lacks the latest security patches, exposing systems to known vulnerabilities. Configuration issues, such as the use of weak encryption methods or default settings, further amplify the risk. Regular audits and proactive monitoring are vital to mitigate these threats.

KQL Queries for Identifying Outdated Software

Using KQL, you can analyze logs and telemetry data to pinpoint systems running outdated versions of software or possessing misconfigurations. Here's a sample query:

```
DeviceSoftwareInventory
```

```
| where SoftwareVersion !contains "latest"
```

```
| summarize OutdatedCount = count() by  
DeviceName  
  
| project DeviceName, OutdatedCount
```

This query filters devices based on software versions and provides a summary of systems that require updates. The use of fields such as DeviceSoftwareInventory depends on your specific environment and data schema, so adapting the query to your setup is essential.

Detecting Vulnerable Configurations

KQL also enables you to search for configuration-related issues. For instance, you can detect systems using outdated encryption algorithms:

Query:

```
DeviceFileCertificateInfo
```

```
| where EncryptionType == 'SHA1'
```

```
| summarize VulnerableDevices = count() by  
DeviceName  
  
| project DeviceName, VulnerableDevices
```

Such queries help identify devices that require immediate configuration changes to align with modern security standards.

Detecting Privilege Escalation and Improper Administrative Access

Privilege escalation is a common attack vector that allows attackers to gain unauthorized access to critical system components. Detecting these attempts is crucial to prevent breaches.

Understanding Privilege Escalation Risks

Privilege escalation exploits can result from improper administrative access, exploitation of bugs, or lateral movement across networked

systems. Monitoring access patterns and flagging anomalies are key to managing these risks.

KQL Query for Privilege Escalation Attempts

KQL can be harnessed to analyze user activity and detect suspicious patterns indicative of privilege escalation. Consider the following query:

```
SecurityEvent  
  
| where EventID == 4672  
  
| extend UserActivity = strcat(SubjectUserName,  
': ', Activity)  
  
| summarize EscalationAttempts = count() by  
UserActivity  
  
| project UserActivity, EscalationAttempts
```

This query identifies events where special privileges were assigned to accounts, flagging potential privilege escalation attempts for further investigation.

Detecting Improper Administrative Access

Sometimes, administrative privileges may be improperly assigned to users, elevating their access rights beyond necessity. To detect such cases, try this KQL query:

```
SecurityEvent
```

```
| where EventID == 4728 or EventID == 4732  
  
| extend AdminChanges = strcat(SubjectUserName,  
' : ', TargetAccount)  
  
| summarize ChangesCount = count() by  
AdminChanges
```

```
| project AdminChanges, ChangesCount
```

This query monitors changes to administrative groups and flags instances where users have been added or removed, allowing you to verify the legitimacy of such modifications.

KQL empowers security teams by providing robust insights into system vulnerabilities and suspicious activities. By crafting precise queries tailored to your environment, you can proactively identify outdated software, misconfigurations, and improper privilege escalations. Integrating these practices into your security protocols ensures you stay ahead of potential threats, enhancing your organization's resilience against cyberattacks.

Next Steps

1. Regularly audit system logs using KQL queries.
2. Customize queries to address your organization's specific vulnerabilities.
3. Automate alert generation for critical findings to ensure swift responses.

Using KQL is not just about detecting vulnerabilities—it's about fostering a proactive, data-driven approach to cybersecurity. Start leveraging its power today to safeguard your systems effectively.

Chapter 33

Using KQL to Enhance Threat Detection

Where cyber threats are evolving at an unprecedented pace, organizations must arm themselves with robust mechanisms to detect and respond to suspicious activities. KQL (Kusto Query Language), designed for querying structured data, has become an invaluable tool for cybersecurity experts. Its ability to sift through vast datasets and pinpoint anomalies makes it a cornerstone in threat detection and response strategies.

This chapter post will guide you through two critical aspects of using KQL for threat detection: revealing anomalous activities in specific geographic locations or IP ranges and filtering events involving known malicious entities.

1. Revealing Anomalous Activities in Specific

Geographic Locations or IP Ranges

Detecting anomalies in geographic locations or IP ranges is vital for identifying potential bad actors or unusual activities. Cyberattacks often originate from unexpected regions or involve unusual patterns related to IP addresses. Here's how you can use KQL to uncover such anomalies:

Sample Query:

```
SigninLogs
```

```
| where Location has_any ("Russia", "China",  
"North Korea")
```

```
| where IPAddress startswith "192.168"
```

```
| summarize Count = count() by Location,  
IPAddress
```

```
| sort by Count desc
```

Explanation:

- SigninLogs: This table contains sign-in records for analysis.
- where Location in ("Russia", "China", "North Korea"): Filters sign-ins originating from specific geographic regions commonly flagged for potential threats.
- IPAddress startswith "192.168": Further narrows down to IP ranges of interest.
- summarize Count = count() by Location, IPAddress: Aggregates and counts activities by location and IP address.
- order by Count desc: Orders results to focus on regions or addresses with the highest activity.

This query enables you to pinpoint regions or IP ranges exhibiting unusual activity levels, which could be a sign of a coordinated attack or suspicious reconnaissance actions.

2. Filtering Events Involving Known Malicious Entities

Another critical element of threat detection is identifying interactions with known malicious entities, such as flagged IP addresses or domains. These indicators of compromise (IoCs) are crucial in understanding and mitigating ongoing threats. Here's how KQL can help:

Sample Query:

```
IdentityQueryEvents
```

```
| where IPAddress in ("203.0.113.0",
"198.51.100.0") or QueryTarget in ("malicious-
domain.com", "phishing-site.org")
| project TimeGenerated, IPAddress,
QueryTarget, ActionType, AccountName
| order by TimeGenerated desc
```

Explanation:

- `IdentityQueryEvents`: This table holds security events for analysis.

- where IPAddress in (...): Filters the dataset for events involving flagged IP addresses.
- DomainName in (...): Captures interactions with flagged domains.
- project: Selects specific columns for streamlined analysis.
- order by Timestamp desc: Sorts events chronologically to prioritize the latest incidents.

This query offers a precise way to focus on events tied to known threats, enabling faster and more efficient incident response.

Best Practices for Using KQL in Threat Detection

To maximize the effectiveness of KQL for threat detection, consider the following tips:

- **Maintain Up-to-Date IoC Lists:** Regularly update your flagged IPs, domains, and other IoCs to ensure your queries are relevant.
- **Leverage Advanced Analytics:** Use advanced KQL functions like anomaly

detection operators to identify subtle deviations over time.

- **Integrate Context:** Combine KQL queries with data enrichment to provide context, such as identifying whether an IP address is part of a known botnet.

KQL's versatility and power make it an essential ally in the battle against cyber threats. Whether you're investigating anomalous activities in specific geographic locations or filtering for events involving malicious entities, mastering KQL can elevate your threat detection capabilities. By employing targeted queries like those discussed in this post, you can stay one step ahead of adversaries and protect your organization from emerging dangers.

Start leveraging the power of KQL today and transform your cybersecurity operations into a proactive and efficient defense mechanism.

Chapter 34

Using KQL to Identify Suspicious Behavior

Identifying and addressing suspicious behavior is critical to protecting sensitive data and organizational assets. The Kusto Query Language (KQL) is a powerful tool that enables security analysts to query large datasets effectively, uncovering patterns of potentially malicious activity. In this post, we'll explore how to use KQL to query for unusual login attempts or failed logins across devices, as well as identify patterns that may indicate potential data exfiltration or unauthorized file access.

NOTE: The following queries are provided as examples only. They are provided to show methods and procedures. Some of the tables may not actually exist.

Querying for Unusual Login Attempts or Failed Logins Across Devices

Unusual login attempts or repeated failed logins are often early indicators of malicious activity, such as brute-force attacks or compromised accounts. KQL provides a structured way to identify these anomalies by analyzing logs from authentication systems.

Basic Query for Failed Logins

To track failed login attempts across devices, you can query the logs for events where the login status indicates failure. Here's an example:

```
SigninLogs
```

```
| where ResultType != 0 // Filter out successful logins
```

```
| summarize FailedAttempts = count() by UserPrincipalName, bin(TimeGenerated, 1h)
```

```
| order by FailedAttempts desc
```

This query:

- Filters out successful login attempts (`ResultType != 0`).
- Groups the results by user, device, and hourly timestamps.
- Counts the number of failed login attempts and sorts them in descending order to highlight the most concerning cases.

Identifying Unusual Login Locations

Attackers often log in from unusual geographic locations. You can use KQL to identify logins from unexpected regions:

SigninLogs

```
| extend LoginLocation =  
strcat(LocationDetails.country, ' - ',  
LocationDetails.state, ', ',  
LocationDetails.city)
```

```
| summarize Count = count() by  
UserPrincipalName, LoginLocation
```

```
| where Count > 1  
  
| order by Count desc
```

This query extends the dataset by adding a "LoginLocation" field and aggregates login attempts by user and location. Analysts can review the results to spot inconsistencies with known travel patterns or home-office settings.

Detecting Impossible Travel Scenarios

To identify impossible travel scenarios—logins from two distant geolocations within an unreasonably short time—you can use a query like this:

SigninLogs

```
| summarize LoginTimes =  
make_set(TimeGenerated, 2), Locations =  
make_set(LocationDetails, 2) by  
UserPrincipalName
```

```
| where array_length(LoginTimes) == 2  
  
| extend TimeDiff = datetime_diff('minute',  
todatetime(LoginTimes[0]),  
todatetime(LoginTimes[1]))  
  
| where TimeDiff < 60  
  
| where Locations[0] != Locations[1]
```

This query analyzes login timestamps and locations for each user, highlighting cases where two consecutive logins occur within an hour but from different locations.

Identifying Patterns of Data Exfiltration or Unauthorized File Access

Detecting data exfiltration or unauthorized file access requires analyzing file activity logs for unusual patterns. These patterns may include

large file downloads, access to sensitive files by unusual users, or spikes in file access activity.

Tracking Large File Downloads

To identify large file downloads, use a query that targets specific thresholds for file size:

```
DeviceFileEvents
```

```
| where ActionType == "FileDownloaded" and  
FileSize > 10000000 // Files larger than 10 MB  
  
| summarize DownloadCount = count() by  
InitiatingProcessAccountUpn, FolderPath  
  
| order by DownloadCount desc
```

This query highlights users and file paths associated with large downloads. It can reveal potential cases of data exfiltration.

Unusual Access to Sensitive Files

For sensitive files, you can create a query to detect unusual users accessing them:

```
DeviceFileEvents
```

```
| where FolderPath has_any ("confidential",
"sensitive", "financial")  
  
| summarize AccessCount = count() by  
InitiatingProcessAccountUpn, FolderPath  
  
| order by AccessCount desc
```

This query filters events for files containing key terms (e.g., "confidential" or "financial") in their names and summarizes access patterns by user.

Detecting Spikes in File Activity

Unusual spikes in file access activity often precede data exfiltration attempts. Here's how to query for this pattern:

```
DeviceFileEvents  
  
| where ActionType in ("FileAccessed",  
"FileModified")  
  
| summarize ActivityCount = count() by  
bin(Timestamp, 1h), InitiatingProcessAccountUpn  
  
| where ActivityCount > 100  
  
| order by ActivityCount desc
```

This query aggregates file activity by user and hourly timestamps, flagging instances where the number of file actions exceeds a preset threshold.

KQL empowers security analysts to proactively monitor and investigate suspicious behavior across an organization's IT environment. By querying for unusual login attempts, failed logins, and patterns of data exfiltration or unauthorized

file access, organizations can detect and mitigate threats early.

The examples provided here offer a starting point for leveraging KQL to enhance your security monitoring. Tailor these queries to your organization's specific needs and log sources for optimal results. In the ongoing battle against cyber threats, understanding how to harness tools like KQL is a vital step toward safeguarding your digital assets.

Chapter 35

KQL for DevOps: Monitoring Azure Pipelines with Performance Insights and Failure Diagnostics

Continuous integration and continuous deployment (CI/CD) pipelines are the backbone of delivering reliable software. Azure DevOps provides a robust platform for managing these pipelines, but to truly optimize performance and diagnose failures, you need powerful tools to analyze pipeline logs. Enter **Kusto Query Language (KQL)**, a query language designed for big data analytics in Azure Data Explorer and Azure Monitor. By leveraging KQL, DevOps teams can query Azure Pipelines logs to gain actionable insights into performance bottlenecks and failure patterns.

In this chapter, we'll explore how to use KQL to monitor Azure Pipelines, focusing on extracting performance metrics and diagnosing pipeline failures. We'll walk through practical examples to demonstrate KQL's capabilities and share tips for integrating these queries into your DevOps workflows.

Why KQL for Azure Pipelines?

Azure DevOps generates detailed logs for every pipeline run, capturing events like build durations, task execution times, and error messages. While the Azure DevOps UI provides a high-level view, it can be challenging to aggregate or analyze this data at scale. KQL shines here because it allows you to:

- **Query large datasets:** Efficiently process thousands of pipeline runs.
- **Extract insights:** Identify trends, such as slow tasks or recurring failures.
- **Automate monitoring:** Integrate queries with Azure Monitor for real-time alerts.
- **Customize analysis:** Tailor queries to your team's specific needs.

By connecting Azure DevOps logs to Azure Monitor Logs (via diagnostic settings or custom log ingestion), you can use KQL to unlock deep insights into your CI/CD pipelines.

Setting Up KQL for Azure Pipelines

Before diving into KQL queries, ensure your Azure DevOps logs are accessible in a Log Analytics workspace:

1. Enable Diagnostic Logs in Azure

DevOps:

- In your Azure DevOps project, navigate to **Project Settings > Pipelines > Settings**.
- Enable diagnostic logging to send pipeline logs to a Log Analytics workspace.

2. Configure Log Analytics:

- Create a Log Analytics workspace in Azure.
- Link it to Azure DevOps by configuring the diagnostic settings to forward pipeline logs.

3. Access Logs in Azure Monitor:

- Use the Azure Portal to access your Log Analytics workspace.
- Open the **Logs** section to write KQL queries against the pipeline logs.

Azure DevOps logs are typically stored in tables like `AzureDevOpsAnalytics` or custom tables, depending on your configuration. For this post, we'll assume logs are in a table called `PipelineLogs`.

KQL in Action: Querying Azure Pipelines Logs

Let's explore two key scenarios: **performance monitoring** and **failure diagnostics**. We'll use sample KQL queries to illustrate how to extract meaningful insights from pipeline logs.

Scenario 1: Performance Monitoring

To optimize CI/CD pipelines, you need to identify slow tasks, long-running builds, or performance degradation over time. KQL makes it easy to aggregate and analyze execution times.

Query 1: Average Build Duration by Pipeline

This query calculates the average duration of pipeline runs for each pipeline over the last 30 days.

```
PipelineLogs
```

```
| where TimeGenerated > ago(30d)
```

```
| where EventType == "BuildCompleted"  
  
| summarize AvgDuration = avg(DurationMs) by  
PipelineName  
  
| order by AvgDuration desc
```

Explanation:

- TimeGenerated > ago(30d): Filters logs from the last 30 days.
- EventType == "BuildCompleted": Targets completed build events.
- summarize AvgDuration = avg(DurationMs): Computes the average duration in milliseconds for each pipeline.
- order by AvgDuration desc: Sorts results to highlight the slowest pipelines.

Use Case: Identify pipelines that consistently take longer to complete, indicating potential optimization opportunities.

Query 2: Slowest Tasks in a Pipeline

This query finds the tasks with the longest average execution time in a specific pipeline.

```
PipelineLogs
```

```
| where TimeGenerated > ago(7d)
```

```
| where PipelineName == "MyApp-CI" and  
EventType == "TaskCompleted"
```

```
| summarize AvgTaskDuration =  
avg(TaskDurationMs) by TaskName
```

```
| order by AvgTaskDuration desc
```

```
| top 5 by AvgTaskDuration
```

Explanation:

- PipelineName == "MyApp-CI": Filters for a specific pipeline.
- EventType == "TaskCompleted": Focuses on task-level events.
- top 5 by AvgTaskDuration: Returns the five slowest tasks.

Use Case: Pinpoint tasks (e.g., unit tests or package restores) that are slowing down your pipeline, so you can optimize or parallelize them.

Query 3: Build Duration Trends Over Time

This query visualizes how build durations have changed over the last 30 days.

```
PipelineLogs
```

```
| where TimeGenerated > ago(30d)
```

```
| where EventType == "BuildCompleted" and  
PipelineName == "MyApp-CI"
```

```
| summarize AvgDuration = avg(DurationMs) by  
bin(TimeGenerated, 1d)
```

```
| render timechart
```

Explanation:

- bin(TimeGenerated, 1d): Groups data by day.
- render timechart: Creates a time-series chart in Azure Monitor.

Use Case: Detect performance degradation, such as increased build times after introducing new tasks or dependencies.

Scenario 2: Failure Diagnostics

When pipelines fail, you need to quickly identify the root cause. KQL can help you analyze error messages, failure frequencies, and affected components.

Query 4: Most Common Failure Reasons

This query identifies the most frequent error messages in failed pipeline runs.

PipelineLogs

```
| where TimeGenerated > ago(7d)
```

```
| where BuildResult == "Failed"
```

```
| summarize FailureCount = count() by  
ErrorMessage  
  
| order by FailureCount desc  
  
  
| top 10 by FailureCount
```

Explanation:

- BuildResult == "Failed": Filters for failed builds.
- summarize FailureCount = count() by ErrorMessage: Counts occurrences of each error message.
- top 10 by FailureCount: Shows the top 10 errors.

Use Case: Identify recurring issues, such as missing dependencies or authentication errors, to prioritize fixes.

Query 5: Failures by Task and Pipeline

This query breaks down failures by task and pipeline to pinpoint problematic areas.

PipelineLogs

```
| where TimeGenerated > ago(7d)

| where EventType == "TaskFailed"

| summarize FailureCount = count() by
PipelineName, TaskName

| order by FailureCount desc
```

Explanation:

- `EventType == "TaskFailed"`: Targets failed tasks.
- `summarize FailureCount = count() by PipelineName, TaskName`: Groups failures by pipeline and task.

Use Case: Identify tasks that fail frequently, such as a specific test suite or deployment step, to focus debugging efforts.

Query 6: Correlating Failures with Code Changes

This query links failed builds to recent code commits to identify potential culprits.

```
PipelineLogs
```

```
| where TimeGenerated > ago(7d)
```

```
| where BuildResult == "Failed"
```

```
| project BuildId, CommitId, ErrorMessage
```

```
| join kind=inner (
```

```
PipelineLogs
```

```
| where EventType == "CodeCommit"
```

```
| project BuildId, CommitId, CommitMessage
```

```
) on BuildId
```

```
| summarize by CommitId, CommitMessage,  
ErrorMessage
```

Explanation:

- join kind=inner: Matches failed builds with their associated code commits.
- summarize by CommitId, CommitMessage, ErrorMessage: Groups results by commit and error.

Use Case: Trace failures back to specific code changes, helping developers fix issues faster.

Visualizing and Automating Insights

To make these insights actionable, integrate KQL queries into your DevOps workflows:

1. Dashboards:

- Use Azure Dashboards to visualize query results (e.g., time charts for build durations or tables for failure counts).
- Pin queries like the ones above to a shared team dashboard for real-time monitoring.

2. Alerts:

- Set up Azure Monitor alerts based on KQL queries. For

example, trigger an alert if the failure rate exceeds a threshold:

```
PipelineLogs
```

```
| where TimeGenerated > ago(1h)
```

```
| summarize FailureRate = countif(BuildResult == "Failed") / count() * 100
```

```
| where FailureRate > 10
```

3. Automation:

- Use Azure Logic Apps to automate responses, such as notifying the team via Teams or creating work items in Azure Boards when failures are detected.

Best Practices for KQL and Azure Pipelines

- **Optimize Queries:** Use filters like TimeGenerated and specific EventType values to reduce query scope and improve performance.

- **Standardize Log Data:** Ensure consistent log formats in Azure DevOps to simplify querying.
- **Iterate on Queries:** Start with broad queries to explore data, then refine them for specific use cases.
- **Document Insights:** Share KQL queries and dashboards with your team to foster a data-driven DevOps culture.

KQL is a game-changer for DevOps teams looking to monitor and optimize Azure Pipelines. By querying pipeline logs, you can uncover performance bottlenecks, diagnose failures, and drive continuous improvement in your CI/CD processes. The examples in this post—ranging from analyzing build durations to correlating failures with code changes—demonstrate KQL's flexibility and power.

Start small by setting up a Log Analytics workspace and experimenting with these queries. As you gain confidence, integrate KQL into dashboards, alerts, and automation to create a proactive monitoring system. With KQL and Azure Pipelines, you'll be well-equipped to deliver faster, more reliable software.

Chapter 36

KQL vs. SQL: Bridging the Gap for Azure Users

In the world of data querying, two powerful languages often come into play for Azure users: **Kusto Query Language (KQL)** and **Structured Query Language (SQL)**. While both are designed to extract insights from data, they serve distinct purposes and excel in different scenarios. This chapter compares KQL and SQL, with a focus on KQL's unique strengths in time-series and unstructured data analysis, to help Azure users choose the right tool for their needs.

What Are KQL and SQL?

- **SQL (Structured Query Language):** The industry-standard language for querying and managing relational databases. SQL is widely used in Azure SQL Database, Azure Synapse Analytics, and other relational database systems. It's optimized for

structured data stored in tables with predefined schemas.

- **KQL (Kusto Query Language):** A query language developed by Microsoft for Azure Data Explorer (ADX) and other Azure services like Azure Monitor and Microsoft Sentinel. KQL is tailored for analyzing large volumes of structured, semi-structured, and unstructured data, with a focus on time-series and log analytics.

While SQL is a general-purpose querying language, KQL is purpose-built for telemetry, logs, and time-series data, making it a go-to choice for Azure's analytics platforms.

Key Differences Between KQL and SQL

Feature	SQL	KQL
Primary Use Case	Relational database management and querying	Log analytics, time-series data, and telemetry analysis
Data Types	Structured data (tables with fixed schemas)	Structured, semi-structured, and unstructured data (e.g., JSON, logs)
Performance	Optimized for joins and transactional queries	Optimized for high-speed, large-scale data ingestion and analysis
Syntax	Standardized, verbose (e.g., <code>SELECT</code> , <code>JOIN</code> , <code>GROUP BY</code>)	Concise, intuitive (e.g., <code>where</code> , <code>summarize</code> , <code>project</code>)
Time-Series Support	Limited; requires complex queries for time-based analysis	Native support with operators like <code>time()</code> , <code>bin()</code> , and <code>series</code>
Learning Curve	Steeper for complex queries; widely known	Easier to learn, especially for analytics-focused tasks
Azure Integration	Azure SQL, Synapse, Cosmos DB	Azure Data Explorer, Azure Monitor, Sentinel

KQL's Strengths in Time-Series and Unstructured Data Analysis

KQL shines in scenarios where SQL may require more effort or customization. Here's why Azure users are increasingly turning to KQL for specific use cases:

1. Native Time-Series Analysis

Time-series data—such as logs, metrics, or IoT telemetry—is central to modern analytics. KQL is designed with time-series in mind, offering built-in operators that simplify temporal analysis:

- **Time Filters:** KQL's where clause integrates seamlessly with time-based conditions, like `where TimeGenerated > ago(1h)` to filter data from the last hour.
- **Binning:** The `bin()` function groups timestamps into intervals (e.g., 5-minute or 1-hour buckets) for aggregation, making it easy to analyze trends.
- **Series Analysis:** KQL supports advanced time-series operations, such as `make-series` to create time-series arrays and `series_fit_line()` for trend analysis.

Example KQL Query for Time-Series:

StormEvents

```
| where StartTime > datetime(2023-01-01) and  
StartTime < datetime(2023-12-31)
```

```
| summarize EventCount = count() by  
bin(StartTime, 1d)
```

```
| render timechart
```

This query aggregates storm events by day and visualizes them as a time chart—something that would require more complex SQL syntax.

In contrast, SQL lacks native time-series functions, often requiring subqueries or window functions to achieve similar results, which can be cumbersome and less performant for large datasets.

2. Handling Unstructured and Semi-Structured Data

KQL excels at querying unstructured or semi-structured data, such as JSON logs or telemetry data, which are common in Azure Monitor or Sentinel. Its dynamic data type and operators like parse, extractjson, and bag_unpack make it easy to work with nested or schema-less data.

Example KQL Query for Unstructured Data:

```
SecurityEvent
```

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

```
| extend ParsedData =  
parse_json(LogonInformation)
```

```
| project Account = ParsedData.AccountName,  
LogonTime = ParsedData.LogonTime
```

This query extracts fields from a JSON blob in a security log, projecting them as structured

columns. In SQL, parsing JSON is possible (e.g., using OPENJSON in Azure SQL), but it's often more verbose and less intuitive.

3. High Performance for Big Data

KQL is optimized for Azure Data Explorer's distributed architecture, enabling lightning-fast queries over massive datasets. Its column-store design and aggressive caching make it ideal for log and telemetry analysis, where SQL's row-based processing may struggle with scale.

4. Concise and Intuitive Syntax

KQL's pipeline-based syntax (using |) is inspired by PowerShell and is more readable for analytics tasks. For example, a KQL query flows naturally from filtering to aggregation to visualization, reducing the cognitive load compared to SQL's more rigid structure.

SQL Equivalent (Complex):

```
SELECT DATEADD(day, DATEDIFF(day, 0,
StartTime), 0) AS Day, COUNT(*) AS EventCount
FROM StormEvents
```

```
WHERE StartTime BETWEEN '2023-01-01' AND  
'2023-12-31'
```

```
GROUP BY DATEADD(day, DATEDIFF(day, 0,  
StartTime), 0)
```

Compare this to the KQL example above—KQL is shorter and easier to write.

5. Visualization and Integration

KQL integrates seamlessly with Azure's visualization tools, like ADX dashboards and Power BI. The render operator allows users to generate charts directly from queries, a feature SQL lacks natively.

When to Use SQL Over KQL

While KQL is powerful for analytics, SQL remains the better choice for:

- **Transactional Workloads:** SQL is ideal for CRUD operations in relational databases, such as updating customer records or managing inventory.
- **Complex Joins:** SQL's mature join capabilities are better suited for combining multiple tables with complex relationships.

- **Standardized Ecosystems:** SQL's universal adoption makes it the default for cross-platform applications or traditional BI tools.

Bridging the Gap: Using KQL and SQL Together in Azure

Azure users don't have to choose one over the other—KQL and SQL complement each other. For example:

- **Hybrid Workflows:** Use SQL in Azure Synapse for data warehousing and ETL, then pipe processed data to Azure Data Explorer for KQL-based log or time-series analysis.
- **Cross-Querying:** Azure Data Explorer supports SQL-like queries, and tools like Azure Synapse allow KQL queries via serverless pools, enabling interoperability.
- **Unified Monitoring:** Azure Monitor combines SQL-based metrics (from Azure SQL) with KQL-based log analytics, providing a holistic view of application performance.

For Azure users, the choice between KQL and SQL depends on the use case. SQL is the go-to for structured, transactional data and relational database management. However, KQL's strengths in **time-series analysis, unstructured data processing, and high-performance analytics** make it indispensable for modern telemetry and log-driven workloads.

By understanding the strengths of both languages, Azure users can leverage KQL for real-time insights and SQL for structured data management, bridging the gap between operational and analytical needs. Whether you're analyzing IoT streams, security logs, or application metrics, KQL's simplicity and power make it a game-changer in the Azure ecosystem.

Ready to dive into KQL? Start exploring with Azure Data Explorer's free tier or try sample queries in the Azure Portal's Log Analytics workspace. For SQL users, consider experimenting with KQL in Azure Monitor to see how it can enhance your analytics workflows.

Chapter 37

From SQL to KQL: A Smooth Transition for Database Pros

As a database professional, you've likely spent years mastering SQL, the lingua franca of relational databases. But what happens when you encounter Kusto Query Language (KQL), the powerful query language behind Azure Data Explorer, Application Insights, and Log Analytics? At first glance, KQL might seem like a distant cousin to SQL—familiar yet distinct. The good news? The transition is smoother than you might think. In this post, we'll bridge the gap by comparing and contrasting SQL and KQL, highlighting key similarities and unique features, complete with side-by-side query examples. Whether you're analyzing logs, telemetry data, or time-series information, understanding these parallels will accelerate your KQL proficiency.

Why Make the Switch? A Quick Primer on KQL

KQL is designed for querying large-scale, semi-structured data in distributed systems. Unlike SQL's focus on relational tables with strict schemas, KQL excels in handling dynamic datasets like JSON logs or event streams. It's optimized for speed and scalability, making it ideal for big data scenarios in Azure environments. If you're coming from SQL Server, Oracle, or PostgreSQL, you'll appreciate KQL's declarative style, but you'll need to adapt to its pipeline-based syntax and specialized operators.

The core philosophy: SQL is statement-oriented, while KQL uses a fluent, pipe-separated flow (inspired by languages like PowerShell or LINQ). This makes queries read like a series of transformations on data.

Key Similarities: Building on Your SQL Foundation

Many SQL concepts translate directly to KQL, easing the learning curve. Let's start with the basics.

1. Selecting Data

Both languages use SELECT to project columns, but KQL often implies it via the table name and pipe (|) for further operations.

- **SQL Example:**

```
SELECT Name, Age FROM Users;
```

- **KQL Equivalent:**

```
Users
```

```
| project Name, Age
```

Here, project in KQL is akin to SQL's SELECT, allowing you to rename or compute columns on the fly.

2. Filtering with WHERE Clauses

The WHERE clause is nearly identical in both, filtering rows based on conditions.

- **SQL Example:**

```
SELECT * FROM Users WHERE Age > 30;
```

- **KQL Equivalent:**

```
Users
```

```
| where Age > 30
```

KQL's `where` is case-insensitive and supports dynamic types, but the logic is the same—predicate-based filtering.

3. Joining Tables

Joins work similarly, though KQL uses `join` with flavors like `inner`, `leftouter`, etc., and emphasizes performance in large datasets.

- **SQL Example:**

```
SELECT u.Name, o.OrderDate
```

```
FROM Users u
```

```
INNER JOIN Orders o ON u.Id = o.UserId;
```

- **KQL Equivalent:**

```
Users
```

```
| join kind=inner Orders on $left.Id ==  
$right.UserId  
  
| project Name, OrderDate
```

Note the \$left and \$right aliases for clarity in KQL, which avoids ambiguity in complex joins.

4. Grouping and Aggregating

Aggregation functions like COUNT, SUM, and AVG are shared, with GROUP BY mirrored by KQL's summarize.

- **SQL Example:**

```
SELECT Department, COUNT(*) AS EmployeeCount  
  
FROM Employees
```

```
GROUP BY Department;
```

- **KQL Equivalent:**

Employees

```
| summarize EmployeeCount = count() by  
Department
```

KQL's `summarize` is more flexible, allowing multiple aggregations and even custom bins for time-series data.

These similarities mean you can often "SQL-ify" your KQL queries initially, then refine them for optimization.

Unique KQL Features: What Sets It Apart

While SQL is great for transactions and ACID compliance, KQL shines in exploratory analysis and pattern detection. Here are standout features that go beyond SQL.

1. The Pipeline Operator (|)

KQL's queries flow through pipes, chaining operations sequentially. This modular approach makes complex queries easier to read and debug—no nested subqueries required.

- **SQL Example (Nested):**

```
SELECT AVG(Age) FROM (SELECT * FROM Users WHERE Country = 'USA');
```

- **KQL Equivalent (Piped):**

```
Users
```

```
| where Country == "USA"  
|  
| summarize AvgAge = avg(Age)
```

Pipes encourage a step-by-step mindset, perfect for data pipelines.

2. Let Statements for Variables and Reusability

KQL's let allows defining variables, functions, or subqueries for reuse—think SQL variables or CTEs, but more lightweight.

- **SQL Equivalent (Using CTE):**

```
WITH ActiveUsers AS (
    SELECT * FROM Users WHERE Status = 'Active'
)

SELECT COUNT(*) FROM ActiveUsers;
```

- **KQL Example:**

```
let ActiveUsers = Users | where Status == "Active";
ActiveUsers | count
```

let promotes cleaner, modular code, especially in long queries. You can even define scalar variables: let threshold = 30;

3. Time-Series and Pattern Matching

KQL has built-in functions for time-based analysis, like bin() for bucketing timestamps, or make-series for creating sequences—features that require custom SQL hacks.

- SQL Example (Basic Time Grouping):**

```
SELECT DATE_TRUNC('day', Timestamp) AS Day,  
COUNT(*)  
FROM Logs
```

```
GROUP BY Day;
```

- KQL Equivalent (With Binning):**

```
Logs
```

```
| summarize Count = count() by bin(Timestamp,  
1d)
```

For patterns, KQL's parse and extract handle semi-structured data effortlessly, far surpassing SQL's string functions.

4. Extensibility with Plugins and Functions

KQL supports user-defined functions (UDFs) and plugins for machine learning (e.g., anomaly detection) or geospatial queries, integrating seamlessly without external tools.

Side-by-Side Query Showdown: Real-World Scenarios

Let's apply these concepts to practical examples. Assume we have a Logs table with columns: Timestamp, UserId, EventType, Duration.

Scenario 1: Basic Filtering and Projection

- **SQL:**

```
SELECT UserId, EventType
```

```
FROM Logs
```

```
WHERE Duration > 100;
```

- **KQL:**

```
Logs
```

```
| where Duration > 100
```

```
| project UserId, EventType
```

Scenario 2: Aggregation with Conditions

- **SQL:**

```
SELECT EventType, AVG(Duration) AS AvgDuration
```

```
FROM Logs
```

```
WHERE Timestamp >= '2023-01-01'
```

```
GROUP BY EventType  
  
HAVING AVG(Duration) > 50;
```

- **KQL:**

Logs

```
| where Timestamp >= datetime(2023-01-01)  
  
| summarize AvgDuration = avg(Duration) by  
EventType  
  
| where AvgDuration > 50
```

Note KQL's post-aggregation where replaces SQL's HAVING.

Scenario 3: Complex Query with Variables and Joins

- **SQL (With CTE):**

```
WITH RecentLogs AS (
    SELECT * FROM Logs WHERE Timestamp >
CURRENT_DATE - INTERVAL '7' DAY
)
SELECT l.UserId, u.Name, COUNT(*) AS EventCount
FROM RecentLogs l
JOIN Users u ON l.UserId = u.Id
GROUP BY l.UserId, u.Name;
```

- **KQL (With Let):**

```
let RecentLogs = Logs | where Timestamp >
ago(7d);
```

RecentLogs

```
| join kind=inner Users on $left.UserId ==  
$right.Id
```

```
| summarize EventCount = count() by UserId,  
Name
```

KQL's `ago(7d)` is a handy relative time function, simplifying date calculations.

Overcoming Common Pitfalls in Transition

- **Case Sensitivity:** KQL is generally case-insensitive for keywords, but column names are case-sensitive—unlike some SQL dialects.
- **No Transactions:** KQL is read-only; focus on queries, not updates.
- **Dynamic Typing:** Embrace flexibility, but watch for type mismatches in comparisons.
- **Performance Tips:** Use `take` for sampling large datasets, and leverage materialized views for frequent queries.

Embrace the Flow

Transitioning from SQL to KQL isn't about unlearning—it's about expanding your toolkit. The similarities provide a safety net, while KQL's unique features like pipelines and let statements unlock new efficiencies for big data analytics. Start by rewriting your familiar SQL queries in KQL, experiment in the Azure Data Explorer web UI, and explore Microsoft's documentation for deeper dives.

If you're a database pro eyeing Azure's ecosystem, KQL will feel like a natural evolution. What's your first KQL query going to be?

Chapter 38

Protecting AI Systems: Detecting Common Attacks with KQL in Microsoft Sentinel

Artificial Intelligence (AI) systems are increasingly integral to modern organizations, powering everything from predictive analytics to automated decision-making. However, their growing adoption makes them prime targets for cyberattacks. Common AI attacks, such as data poisoning, model evasion, and model inversion, can compromise model integrity, leak sensitive data, or degrade performance. Using Microsoft Sentinel, security teams can leverage Kusto Query Language (KQL) to detect and mitigate these threats effectively. This chapter explores prevalent AI attacks and provides practical KQL queries using Microsoft Sentinel's `SecurityEvent` and `AzureDiagnostics` tables to identify suspicious activities.

Common AI Attacks

1. **Data Poisoning:** Attackers manipulate training data to bias model outputs, often by injecting malicious inputs or outliers. This can lead to incorrect predictions or degraded performance.
2. **Model Evasion:** Adversaries craft inputs to bypass AI model defenses, such as submitting adversarial examples that trick the model into misclassifying data.
3. **Model Inversion:** Attackers exploit model outputs to reconstruct sensitive training data, compromising privacy.
4. **Model Extraction:** Malicious actors repeatedly query the model to replicate its functionality, stealing intellectual property.
5. **Probing and Brute-Force Attacks:** Attackers send rapid or malformed requests to probe for vulnerabilities or overwhelm the system.

By monitoring logs in Microsoft Sentinel, organizations can detect these attacks early. Below are five KQL query examples tailored to identify these threats using the `SecurityEvent` and `AzureDiagnostics` tables, which are commonly available in Sentinel environments.

KQL Queries for AI Attack Detection

The following queries assume a log structure with fields

like `TimeGenerated, Account, SourceIp, EventID, OperationName, ResultType, request_s, response_s, payloadSize_d, confidenceScore_d, and anomalyScore_d.` Adjust field names and thresholds to match your environment.

1. Detecting Rapid Repeated API Calls (Probing or Brute-Force)

Rapid, repeated requests to AI endpoints may indicate probing or brute-force attacks. This query monitors logon events to AI-related accounts or APIs.

```
// Detect rapid repeated API calls to AI endpoints
```

```
SecurityEvent
```

```
| where EventID == 4624 or EventID == 4672 // Logon events
```

```
| where TargetAccount contains "AI" or  
LogonProcessName contains "api"  
  
| summarize RequestCount = count() by SourceIp,  
bin(TimeGenerated, 1m)  
  
| where RequestCount > 50  
  
| project TimeGenerated, SourceIp, RequestCount
```

Why it works: This query flags IPs making over 50 requests per minute, which is unusual for legitimate users and may indicate automated probing.

2. Identifying High-Confidence Model Outputs from Suspicious IPs

Adversarial inputs often produce unusually high-confidence outputs. This query detects such patterns from uncommon IPs.

```
// Identify high-confidence AI model outputs  
from unusual IPs
```

```
AzureDiagnostics
```

```
| where OperationName == "InferenceRequest"  
  
| where confidenceScore_d > 0.99  
  
| summarize HighConfidenceCount = count() by  
SourceIp, Account  
  
| where HighConfidenceCount > 10  
  
| project TimeGenerated, SourceIp, Account,  
HighConfidenceCount
```

Why it works: High confidence scores (>0.99) from a single IP with many requests suggest potential adversarial inputs designed to exploit the model.

3. Detecting Data Poisoning via Large Payloads

Data poisoning often involves submitting oversized or outlier inputs. This query identifies unusually large payloads in training data uploads.

```
// Detect potential data poisoning via large input payloads

AzureDiagnostics

| where OperationName == "TrainingDataUpload"

| where payloadSize_d > avg(payloadSize_d) + 3 * stdev(payloadSize_d)

| project TimeGenerated, Account,
payloadSize_d, requestUri_s
```

Why it works: Payloads exceeding three standard deviations from the average size are flagged as potential poisoning attempts.

4. Flagging Model Inversion via Sensitive Data in Outputs

Model inversion attacks aim to extract sensitive data from model outputs. This query searches for sensitive patterns, like Social Security Numbers, in responses.

```
// Detect model inversion attempts by monitoring sensitive data

AzureDiagnostics
| where OperationName == "InferenceRequest"
| where response_s contains "sensitive" or response_s matches regex "[0-9]{3}-[0-9]{2}-[0-9]{4}"
| project TimeGenerated, Account, response_s, SourceIp
```

Why it works: Detecting sensitive data patterns in model outputs indicates a potential breach of training data privacy.

5. Detecting Model Extraction via High-Frequency Unique Queries

Model extraction involves querying the model repeatedly to replicate its behavior. This query identifies accounts with a high number of unique requests.

```
// Detect model extraction via high-frequency unique queries
```

```
AzureDiagnostics
```

```
| where OperationName == "InferenceRequest"
```

```
| summarize UniqueQueries = dcount(request_s)  
by Account, bin(TimeGenerated, 1h)
```

```
| where UniqueQueries > 50
```

```
| project TimeGenerated, Account, UniqueQueries
```

Why it works: A high count of unique queries within an hour suggests an attempt to systematically map the model's behavior.

Implementing These Queries in Microsoft Sentinel

To use these queries:

1. **Ingest Logs:** Ensure AI system logs (e.g., inference requests, training data uploads) are ingested into Sentinel via Azure Diagnostics or custom connectors.
2. **Create Alerts:** Save these queries as analytics rules in Sentinel, configuring thresholds and schedules based on your environment.
3. **Tune Thresholds:** Adjust thresholds (e.g., `RequestCount > 50`, `confidenceScore_d > 0.99`) to minimize false positives while catching true threats.
4. **Correlate with Other Logs:** Join with `SigninLogs` or `AADNonInteractiveUserSignInLogs` for additional context, such as user authentication patterns.

5. **Respond to Incidents:** Set up automated responses, like blocking IPs or notifying security teams, using Sentinel playbooks.

Best Practices for Securing AI Systems

- **Input Validation:** Sanitize and validate all inputs to AI models to prevent adversarial examples and poisoning.
- **Rate Limiting:** Enforce API rate limits to deter probing and extraction attempts.
- **Anomaly Detection:** Use machine learning in Sentinel to baseline normal behavior and flag deviations.
- **Access Controls:** Restrict access to AI endpoints using Azure AD and monitor unauthorized access attempts.
- **Regular Auditing:** Continuously audit logs for suspicious patterns and update KQL queries as new threats emerge.

AI systems are powerful but vulnerable to sophisticated attacks. By leveraging Microsoft Sentinel's robust logging and KQL's querying capabilities, organizations can proactively detect and respond to threats like data poisoning, model

evasion, and model inversion. The provided KQL queries offer a starting point for building a comprehensive AI security monitoring strategy. Regularly refine these queries and integrate them with Sentinel's alerting and automation features to stay ahead of attackers.

For more advanced setups or custom connectors, refer to Microsoft Sentinel documentation or consult with your security team to tailor these queries to your AI environment.

Chapter 39

Exploring the Limits: What KQL Can't Do—and Creative Workarounds

Kusto Query Language (KQL), the query language powering Azure Data Explorer, Log Analytics, and other Microsoft data platforms, is a powerhouse for slicing through massive datasets with speed and precision. Its intuitive syntax and robust functions make it a go-to for data analysts, engineers, and security professionals. But even KQL has its limits. From missing features to performance quirks, there are times when KQL alone won't cut it. Fear not! In this post, we'll dive into KQL's key limitations and explore *creative*, sometimes offbeat workarounds to get the job done. Buckle up for a wild ride through the KQL wilderness!

KQL's Limitations: Where It Falls Short

Before we get to the fun stuff, let's lay out the main areas where KQL can leave you scratching your head:

1. **No Procedural Programming:** KQL is declarative, not procedural. You can't write loops, conditionals, or stored procedures like in SQL or Python.
2. **Limited String Manipulation:** Advanced string operations (e.g., regex replacements or complex parsing) are clunky or nonexistent.
3. **No Write Operations:** KQL is read-only. You can't update, insert, or delete data directly.
4. **Restricted Joins:** Cross-cluster joins are limited, and joining large datasets can hit performance walls.
5. **No Native Support for Complex Math/Stats:** Advanced statistical models or matrix operations? KQL's not your guy.
6. **Exporting Results:** Getting query results out of KQL for external use is tricky without integration tools.
7. **Dynamic Schema Challenges:** Handling highly variable or nested JSON data can feel like herding cats.

Now, let's push the boundaries with some unconventional workarounds that'll make you rethink what's possible with KQL.

1. No Loops? Fake It with Recursive Queries

Problem: KQL lacks traditional for or while loops, so tasks like iterating over a range or generating sequences are tough. Want to create a time series for missing data points? Good luck without a loop.

Creative Workaround: Use recursive Common Table Expressions (CTEs) with union to simulate loops. KQL doesn't call it recursion, but you can chain queries to build iterative logic.

Example: Let's say you need to generate a sequence of dates to fill gaps in a time-series dataset.

```
let startDate = datetime(2025-01-01);
```

```
let endDate = datetime(2025-01-10);
```

```
let dateSequence = range i from 0 to
toint((endDate - startDate)/1d) step 1

| project GeneratedDate = startDate + i * 1d;

dateSequence

| join kind=leftouter (
    MyTable

    | where EventTime between (startDate ..
endDate)

) on $left.GeneratedDate == $right.EventTime

| project GeneratedDate, EventCount =
coalesce(EventCount, 0)
```

How It Works: The range operator generates a sequence of numbers, which we convert to dates. We then left-join this with your actual data to fill gaps with zeros. It's not a true loop, but it mimics iterative behavior.

Offbeat Twist: Need to “loop” over non-numeric data, like a list of strings? Use mv-expand to explode an array and process each element, chaining multiple mv-expand for nested iterations. For example, to process a dynamic JSON array:

```
let data = datatable(id: string, values: dynamic) [  
    ("a", dynamic(["x", "y", "z"])),  
    ("b", dynamic(["p", "q"]))  
];  
  
data
```

```
| mv-expand value = values to typeof(string)  
  
| project id, value
```

This “explodes” the array into rows, letting you process each element as if you were looping.

2. String Manipulation: Get Hacky with `parse` and `replace`

Problem: KQL’s string functions (split, substring, trim) are basic. No regex replacements or advanced parsing? That’s a bummer for log analysis or text processing.

Creative Workaround: Lean on `parse`, `extract`, and creative use of `replace_string` to approximate regex-like behavior. For complex cases, preprocess data outside KQL and ingest it back.

Example: You need to extract a version number (e.g., v2.3.1) from a messy log string like app-v2.3.1-prod.

```
let logs = datatable(log: string) [ "app-v2.3.1-prod", "app-v1.0.0-dev" ];
```

```
logs
```

```
| parse log with * "v" version:string "-" *
```

```
| project version
```

How It Works: The parse operator matches patterns and extracts the version number. It's not regex, but it's close enough for simple cases.

Offbeat Twist: For really gnarly strings, use extract with a regex pattern or chain multiple replace_string calls to clean up data. If that fails, export the data to a CSV, process it with Python (using pandas or regex), and re-ingest it into Azure Data Explorer. Here's a quick Python snippet to handle complex string parsing:

```
import pandas as pd
```

```
import re
```

```
# Sample data

logs = pd.DataFrame({"log": ["app-v2.3.1-prod",
"app-v1.0.0-dev"] })

# Extract version with regex

logs["version"] = logs["log"].apply(lambda x:
re.search(r"v(\d+\.\d+\.\d+)", x).group(1))

# Export to CSV for re-ingestion

logs.to_csv("cleaned_logs.csv", index=False)
```

Re-ingest the CSV into Kusto and query away. It's a detour, but it works!

3. No Write Operations? Queue Up External Actions

Problem: KQL can't modify data. Need to flag anomalies or update records based on a query? You're stuck.

Creative Workaround: Use KQL to identify records, then trigger external actions via Azure Functions, Logic Apps, or Event Grid. Export results to a queue or storage, and let another system handle the writes.

Example: Flag suspicious login attempts and send them to a queue for processing.

SecurityLogs

```
| where EventType == "Login" and IsSuspicious  
== true  
  
| project UserId, Timestamp, IPAddress  
  
| summarize by UserId, Timestamp, IPAddress
```

How It Works: Pipe the results to an Azure Function via an Event Grid trigger. The function can write to a database or send alerts. Set this up in Azure:

1. Configure an Event Grid topic in Azure Data Explorer.
2. Create an Azure Function to process the events (e.g., write to Cosmos DB).
3. Subscribe the function to the Event Grid topic.

Offbeat Twist: For a low-tech hack, export query results to a Blob Storage CSV and use a scheduled script (e.g., PowerShell or Python) to process the file and update your database. It's not elegant, but it's effective for one-off tasks.

4. Restricted Joins? Shard and Conquer

Problem: Cross-cluster joins are limited, and joining massive tables can choke performance.

Creative Workaround: Break joins into smaller chunks using materialize or pre-aggregate data to reduce cardinality. For cross-cluster scenarios, replicate data or use external tools to merge results.

Example: Joining two large tables across clusters.

```
let T1 = materialize(Cluster1.Database1.Table1  
| where Time > ago(1h) | summarize by  
KeyColumn);
```

```
let T2 = materialize(Cluster2.Database2.Table2  
| where Time > ago(1h) | summarize by  
KeyColumn, Value);
```

T1

```
| join kind=inner T2 on KeyColumn  
  
| project KeyColumn, Value
```

How It Works: materialize caches intermediate results, reducing compute load. Pre-summarizing shrinks the dataset before the join.

Offbeat Twist: If cross-cluster joins are a no-go, export both tables to a common storage (e.g., Data Lake), merge them with Spark or Python, and re-

ingest the result. For example, use Databricks to run:

```
from pyspark.sql import SparkSession

spark =
SparkSession.builder.appName("MergeTables").get
OrCreate()

t1 = spark.read.parquet("dbfs:/t1.parquet")

t2 = spark.read.parquet("dbfs:/t2.parquet")

merged = t1.join(t2, "KeyColumn", "inner")

merged.write("dbfs:/merged.parquet")
```

Re-ingest the merged data into Kusto for further querying.

5. No Complex Math? Outsource to Plugins

Problem: KQL's math functions are basic. Need linear regression, Fourier transforms, or neural nets? You're out of luck.

Creative Workaround: Use KQL's Python or R plugins to run advanced computations within a query. These plugins let you embed scripts directly.

Example: Run a linear regression on sales data.

```
SalesData
```

```
| evaluate python (
```

```
    ...
```

```
from sklearn.linear_model import  
LinearRegression  
  
  
import pandas as pd  
  
  
  
  
X = dataset[['Day']].values  
  
  
  
  
y = dataset['Sales'].values  
  
  
  
  
model = LinearRegression()  
  
  
  
  
model.fit(X, y)  
  
  
  
  
predictions = model.predict(X)  
  
  
  
  
result = pd.DataFrame({'Day': X.flatten(),  
'PredictedSales': predictions})  
  
` `` ,
```

```
[ 'Day', 'Sales' ]  
  
)  
  
)
```

How It Works: The python plugin runs the script, taking the query results as input and outputting predictions. The R plugin works similarly for R fans.

Offbeat Twist: For super-complex math (e.g., deep learning), train a model outside KQL (e.g., in TensorFlow), save the predictions, and ingest them into Kusto as a lookup table. Query the table for real-time scoring.

6. Exporting Results: Sneaky Data Exfiltration

Problem: Exporting KQL results to CSV or other formats is clunky without Power BI or Azure integrations.

Creative Workaround: Use the render operator to generate a table, copy-paste it manually, or automate exports via Azure Data Factory or Logic Apps.

Example: Export a summary table to Blob Storage.

```
MyTable
```

```
| summarize Total = count() by Category
```

```
| export csv to  
'https://mystorage.blob.core.windows.net/output  
/summary.csv'
```

How It Works: The export command (available in some KQL environments) writes directly to Blob Storage. Check your platform's support.

Offbeat Twist: For a guerrilla approach, render the results as a table in the Kusto Web UI, copy-paste into Excel, and save as CSV. For automation, use a browser script (e.g., Selenium) to scrape the UI and save the data. It's hacky, but it works in a pinch.

7. Dynamic Schema Chaos? Tame It with dynamic

Problem: Nested JSON or variable schemas are a pain to query without explicit columns.

Creative Workaround: Use dynamic types with `mv-expand` and `bag_unpack` to flatten JSON. For extreme cases, preprocess schemas outside KQL.

Example: Query a table with nested JSON.

```
let data = datatable(record: dynamic) [  
    dynamic({ "id": 1, "details": { "name": "Alice", "age": 30 } }),  
    dynamic({ "id": 2, "details": { "name": "Bob", "age": 25 } })  
];  
  
data
```

```
| project id = record.id, details =  
record.details  
  
| evaluate bag_unpack(details)
```

How It Works: `bag_unpack` flattens the `details` object into columns (`name`, `age`). Use `mv-expand` for arrays.

Offbeat Twist: If the JSON is too wild, ingest it into a staging table, use a Python script to normalize it, and re-ingest the clean data. For example:

```
import pandas as pd
```

```
import json
```

```
# Sample JSON
```

```
data = [  
        {"id": 1, "details": {"name": "Alice",  
"age": 30}},  
        {"id": 2, "details": {"name": "Bob", "age":  
25}}]  
  
# Normalize  
  
df = pd.json_normalize(data)  
  
df.to_csv("normalized.csv", index=False)
```

Re-ingest the CSV for clean querying.

KQL's Limits Are Just the Start

KQL's limitations might seem like roadblocks, but with a bit of creativity, they're more like speed bumps. By combining KQL's strengths (fast queries, dynamic types, plugins) with external tools (Python, Azure Functions, Databricks), you can tackle almost any data challenge. The key is to think outside the query box—whether it's faking loops with `mv-expand`, outsourcing math to plugins, or sneaking data out via Blob Storage.

So, next time KQL says “can’t,” respond with a smirk and a workaround. What’s the wildest KQL hack you’ve tried? Share your tricks and let’s keep pushing the limits!

Chapter 40

Automating Security Incident Investigation with KQL: Leveraging mv-expand, project, and where for Alert Analysis

Timely and accurate incident investigation is critical to mitigating threats. Security analysts often face overwhelming volumes of alerts, making manual analysis inefficient. Microsoft's Kusto Query Language (KQL), used in platforms like Microsoft Sentinel and Microsoft Defender, offers powerful tools to automate and streamline this process. In this chapter, we'll explore how KQL functions such as mv-expand, project, and where can be used to analyze security alerts efficiently, enabling faster and more precise incident investigations.

Why Automate with KQL?

KQL is designed to handle large datasets with speed and flexibility, making it ideal for security operations. By automating alert analysis, organizations can:

- **Reduce Mean Time to Detect (MTTD):** Quickly identify critical alerts.
- **Improve Mean Time to Respond (MTTR):** Accelerate investigation with structured queries.
- **Minimize Analyst Fatigue:** Automate repetitive tasks, freeing analysts for deeper analysis.

Let's dive into how mv-expand, project, and where can transform your incident investigation workflow.

Scenario: Analyzing Security Alerts

Imagine you're a security analyst tasked with investigating alerts from Microsoft Defender for Endpoint. The alerts are stored in a table called SecurityAlert, which contains details like alert name, timestamp, affected entities, and severity. Some alerts include nested or multi-value fields, such as lists of affected users or devices, which complicates analysis. Your goal is to extract actionable insights, such as identifying high-severity alerts targeting critical assets.

Here's how KQL can help.

Step 1: Filtering Alerts with where

The `where` operator is your first line of defense for narrowing down the dataset. It filters rows based on specified conditions, ensuring you focus on relevant alerts.

Example Query:

```
SecurityAlert
```

```
| where AlertSeverity == "High" and  
TimeGenerated >= ago(24h)
```

Explanation:

- `AlertSeverity == "High"`: Filters for high-severity alerts.
- `TimeGenerated >= ago(24h)`: Limits results to the last 24 hours.
- This reduces the dataset to a manageable subset, focusing on recent, critical alerts.

Use Case: Use `where` to prioritize alerts by severity, time, or specific threat categories (e.g., `AlertName contains "Ransomware"`).

Step 2: Simplifying Output with project

Once you've filtered the data, the project operator helps you select only the columns you need, making the output cleaner and easier to analyze.

Example Query:

```
SecurityAlert
```

```
| where AlertSeverity == "High" and  
TimeGenerated >= ago(24h)
```

```
| project AlertName, TimeGenerated,  
AffectedEntities, AlertSeverity
```

Explanation:

- `project AlertName, TimeGenerated, AffectedEntities, AlertSeverity`: Selects only the specified columns, discarding irrelevant fields like metadata or internal IDs.
- This creates a focused dataset for further analysis or reporting.

Use Case: Use project to tailor outputs for dashboards, reports, or downstream automation tasks, ensuring only essential information is included.

Step 3: Handling Nested Data with mv-expand

Security alerts often include multi-value or nested fields, such as a JSON array of affected users or devices. The mv-expand operator “flattens” these arrays into individual rows, making it easier to analyze each element.

Example Query:

```
SecurityAlert
```

```
| where AlertSeverity == "High" and  
TimeGenerated >= ago(24h)
```

```
| project AlertName, TimeGenerated,  
AffectedEntities, AlertSeverity
```

```
| mv-expand AffectedEntities
```

Explanation:

- mv-expand AffectedEntities: Expands the AffectedEntities array (e.g., ["user1", "user2"]) into separate rows, with each row containing one entity.
- The resulting table has one row per entity, preserving other columns like AlertName and AlertSeverity.

Use Case: Use mv-expand to analyze relationships, such as identifying all users affected by a specific alert or correlating alerts across multiple devices.

Putting It All Together: A Complete Investigation Query

Let's combine these operators into a comprehensive query to investigate high-severity alerts, extract affected entities, and identify critical assets.

Complete Query:

```
SecurityAlert
```

```
| where AlertSeverity == "High" and  
TimeGenerated >= ago(24h)  
  
| project AlertName, TimeGenerated,  
AffectedEntities, AlertSeverity  
  
| mv-expand AffectedEntities  
  
| where AffectedEntities contains "critical-  
server"  
  
| project AlertName, TimeGenerated,  
AffectedEntities, AlertSeverity
```

Explanation:

1. **Filter:** where AlertSeverity == "High" and TimeGenerated >= ago(24h) narrows the dataset to recent, high-severity alerts.
2. **Select:** project reduces the output to relevant columns.
3. **Expand:** mv-expand AffectedEntities flattens the multi-value field.

4. **Refine:** where AffectedEntities contains "critical-server" filters for alerts affecting critical assets.
5. **Output:** project ensures the final output is clean and focused.

Output Example:

AlertName	TimeGenerated	AffectedEntities	AlertSeverity
Suspicious Logon	2025-06-02T07:00:00	critical-server-01	High
Malware Detected	2025-06-02T06:30:00	critical-server-02	High

This output highlights high-severity alerts affecting critical servers, ready for immediate action or reporting.

Advanced Tips for KQL Automation

1. **Summarization:** Use summarize to aggregate data, e.g., count alerts by AlertName or AffectedEntities.

```
| summarize AlertCount = count() by AlertName,  
AffectedEntities
```

1. **Joins:** Combine SecurityAlert with other tables (e.g., DeviceInfo) to enrich alerts with device details.

```
| join kind=inner (DeviceInfo) on  
$left.AffectedEntities == $right.DeviceName
```

1. **Automation:** Save queries as KQL functions or integrate them into Azure Logic Apps for automated workflows, such as triggering notifications for high-severity alerts.
2. **Visualization:** Pipe results to a render operator for charts in Microsoft Sentinel.

```
| render timechart
```

Benefits of This Approach

By leveraging mv-expand, project, and where, you can:

- **Handle Complex Data:** Easily process nested or multi-value fields.
- **Focus on What Matters:** Filter and project only relevant information.
- **Scale Investigations:** Automate repetitive tasks, allowing analysts to focus on strategic response.

This KQL-driven approach empowers security teams to respond faster, with greater precision, to potential threats.

Automating security incident investigation with KQL transforms how teams handle alerts.

Functions like mv-expand, project, and where provide the flexibility to dissect complex datasets, extract actionable insights, and prioritize critical threats. By incorporating these techniques into your workflows, you can enhance your security operations, reduce response times, and stay ahead of cyber threats.

Ready to try it? Fire up Microsoft Sentinel or Microsoft Defender, craft your KQL queries, and start automating your incident investigations today!

Chapter 41

Detecting Suspicious Activities in Microsoft Sentinel: Writing Threat-Hunting Queries Using KQL

Microsoft Sentinel is a powerful cloud-native Security Information and Event Management (SIEM) solution that empowers organizations to detect, investigate, and respond to security threats. One of its core strengths is the ability to perform proactive threat hunting using Kusto Query Language (KQL). By writing effective KQL queries, security analysts can uncover suspicious activities, identify potential threats, and strengthen their organization's security posture. In this chapter, we'll explore how to write threat-hunting queries in Microsoft Sentinel using KQL, with practical examples and best practices.

Why Threat Hunting in Microsoft Sentinel?

Threat hunting is the proactive process of searching for hidden threats or anomalies within

an organization's environment. Unlike reactive incident response, threat hunting assumes that adversaries may already be present and focuses on finding them before they cause harm.

Microsoft Sentinel's integration with Azure Data Explorer and its use of KQL make it an ideal platform for threat hunting, offering:

- **Scalability:** Handle massive volumes of security data from multiple sources.
- **Flexibility:** Write custom queries to detect specific behaviors or patterns.
- **Integration:** Correlate data from Azure, Microsoft 365, and third-party sources.
- **Automation:** Schedule queries or create analytics rules for continuous monitoring.

KQL is a user-friendly, SQL-like query language that allows analysts to filter, aggregate, and analyze log data efficiently. Let's dive into how to craft KQL queries for threat hunting in Microsoft Sentinel.

Getting Started with KQL in Microsoft Sentinel

Before writing queries, familiarize yourself with Microsoft Sentinel's **Logs** blade, where you can

run KQL queries against your data. Key tables for threat hunting include:

- **SecurityEvent**: Windows event logs, including login events.
- **SigninLogs**: Azure Active Directory sign-in activities.
- **OfficeActivity**: Microsoft 365 activities (e.g., SharePoint, Exchange).
- **AzureActivity**: Azure resource management logs.
- **Syslog**: Linux and network device logs.

To access these tables, ensure your data sources are connected to Sentinel and logs are being ingested.

Writing Threat-Hunting Queries: Best Practices

Effective threat-hunting queries are specific, efficient, and actionable. Follow these best practices:

1. **Define Your Hypothesis**: Start with a clear goal. For example, "Detect multiple failed login attempts followed by a successful login," which could indicate brute-force attacks.

2. **Use Filters Early:** Apply filters like where clauses to reduce data volume and improve query performance.
3. **Leverage Joins and Summarization:** Correlate data across tables and aggregate results to identify patterns.
4. **Test and Refine:** Run queries on a small dataset first, then scale up. Adjust thresholds to minimize false positives.
5. **Schedule or Automate:** Convert successful queries into analytics rules or scheduled hunts for ongoing monitoring.

Now, let's explore some practical KQL queries for common threat-hunting scenarios.

Example 1: Detecting Brute-Force Login Attempts

Brute-force attacks involve repeated login attempts to guess credentials. To detect this, we can look for multiple failed logins followed by a successful login from the same IP address.

```
let timeWindow = 1h;
```

```
let failedThreshold = 5;
```

```
SigninLogs
```

```
| where TimeGenerated > ago(timeWindow)  
  
| where ResultType != 0 // Failed logins  
  
| summarize FailedCount = count() by  
UserPrincipalName, IPAddress  
  
| where FailedCount >= failedThreshold  
  
| join kind=inner (
```

```
SigninLogs
```

```
| where TimeGenerated > ago(timeWindow)  
  
| where ResultType == 0 // Successful  
login
```

```
) on UserPrincipalName, IPAddress  
  
| project TimeGenerated, UserPrincipalName,  
IPAddress, FailedCount, ResultDescription
```

Explanation:

- timeWindow and failedThreshold set the time range and minimum failed attempts.
- The first part counts failed logins (ResultType != 0) per user and IP.
- The join correlates with successful logins (ResultType == 0) from the same user and IP.
- The result shows potential brute-force successes for investigation.

Example 2: Identifying Suspicious PowerShell Activity

Adversaries often use PowerShell for malicious activities. This query detects unusual PowerShell command executions in Windows event logs.

```
SecurityEvent
```

```
| where TimeGenerated > ago(1d)

| where EventID == 4688 // Process creation
events

| where ProcessName contains "powershell.exe"

| where CommandLine !contains
"expected_script_name" // Exclude known
scripts

| summarize CommandCount = count(), Commands =
make_set(CommandLine) by Account, Computer

| where CommandCount > 3

| project TimeGenerated, Account, Computer,
CommandCount, Commands
```

Explanation:

- Filters for process creation events (EventID == 4688) involving powershell.exe.
- Excludes known, legitimate scripts to reduce noise.
- Aggregates commands by user and computer, flagging accounts with high PowerShell activity.
- Outputs commands for review, helping identify malicious scripts.

Example 3: Detecting Data Exfiltration via SharePoint Downloads

Data exfiltration often involves unusual file downloads from SharePoint. This query identifies users downloading multiple files in a short period.

OfficeActivity

```
| where TimeGenerated > ago(1h)
```

```
| where Operation == "FileDownloaded" and  
OfficeWorkload == "SharePoint"  
  
| summarize DownloadCount = count(), Files =  
make_set(OfficeObjectId) by UserId, ClientIP  
  
| where DownloadCount > 10  
  
| project TimeGenerated, UserId, ClientIP,  
DownloadCount, Files
```

Explanation:

- Targets SharePoint file download events (Operation == "FileDownloaded").
- Aggregates downloads by user and IP, flagging users with more than 10 downloads in an hour.
- Lists downloaded files for context, aiding investigation of potential exfiltration.

Advanced Techniques

To enhance your threat-hunting queries:

- **Use Machine Learning:** Leverage Sentinel's built-in machine learning functions, like anomalydetection, to identify outliers.
- **Incorporate Threat Intelligence:** Join your queries with threat intelligence feeds (e.g., ThreatIntelligenceIndicator) to detect known malicious IPs or domains.
- **Visualize Results:** Use Sentinel's workbooks to create dashboards for query results, making patterns easier to spot.
- **Chain Behaviors:** Look for sequences of events, like a failed login followed by a privilege escalation, to detect multi-stage attacks.

Converting Queries to Analytics Rules

Once a query proves effective, turn it into an analytics rule for automated alerting:

1. Go to **Analytics** in Sentinel.
2. Create a new rule and paste your KQL query.
3. Set a schedule (e.g., run every hour) and a look-back period.

4. Define alert thresholds and actions (e.g., create an incident or send an email).
5. Test the rule to ensure it generates meaningful alerts.

Threat hunting with KQL in Microsoft Sentinel empowers security teams to proactively uncover suspicious activities and mitigate risks. By crafting targeted queries, leveraging Sentinel's rich data sources, and following best practices, you can detect threats like brute-force attacks, malicious PowerShell activity, or data exfiltration. Start with simple queries, refine them based on your environment, and automate them for continuous monitoring. With practice, KQL can become a cornerstone of your threat-hunting strategy, helping you stay one step ahead of adversaries.

Happy hunting!

Chapter 42

Building Custom Dashboards with KQL & Workbook Queries: Enhancing Visibility in Security Operations

Visibility into your organization's security posture is critical. Security Operations Centers (SOCs) rely on tools that provide actionable insights to detect, investigate, and respond to threats efficiently. Microsoft Sentinel (now part of Microsoft Defender XDR) empowers SOC teams to create custom dashboards using Kusto Query Language (KQL) and Azure Workbooks. These dashboards provide tailored visualizations to monitor security events, identify anomalies, and streamline incident response.

In this chapter, we'll explore how to build custom dashboards with KQL and Workbook queries to enhance visibility in security operations. We'll cover the basics, provide step-by-step instructions, and share practical examples to help you get started.

Why Custom Dashboards Matter in Security Operations

Custom dashboards address the unique needs of your SOC by:

- **Centralizing Insights:** Aggregate data from multiple sources (e.g., logs, alerts, incidents) into a single view.
- **Improving Detection:** Highlight patterns, trends, or anomalies that might indicate a security threat.
- **Streamlining Response:** Provide analysts with quick access to relevant data for faster decision-making.
- **Tailoring Visualizations:** Create charts, graphs, and tables that align with your team's workflows and priorities.

By leveraging KQL and Workbooks, you can query vast amounts of security data and present it in a way that's intuitive and actionable.

Prerequisites

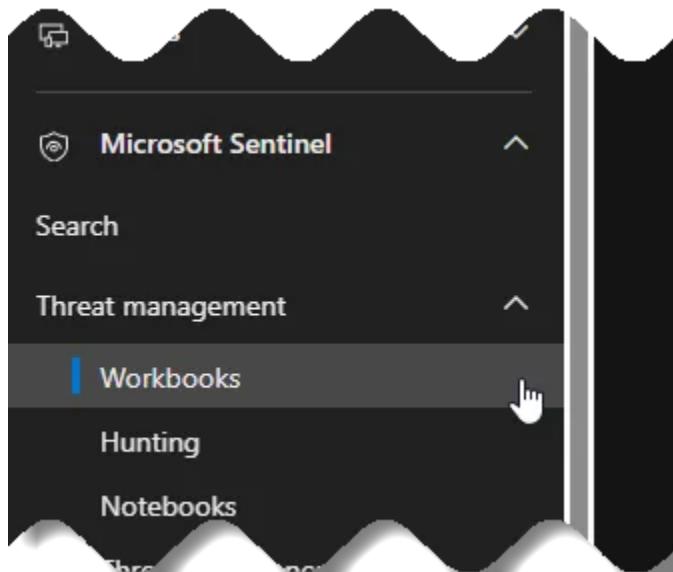
Before diving in, ensure you have:

1. **Access to Microsoft Sentinel/Defender XDR:** You need permissions to create and edit Workbooks.
2. **Basic KQL Knowledge:** Familiarity with KQL syntax is helpful but not mandatory (we'll provide examples).
3. **Data Sources:** Ensure your security data (e.g., Entra logs, Windows event logs, or network traffic) is ingested into Microsoft Sentinel.

Step-by-Step Guide to Building Custom Dashboards

Step 1: Access Workbooks

1. Navigate to **Microsoft Defender XDR** or **Microsoft Sentinel**.
2. In the left-hand menu, select Microsoft Sentinel - Threat Management - **Workbooks**.



3. Click **+ New** to create a new Workbook or choose an existing template to customize.

Step 2: Understand Workbook Components

Workbooks support various elements:

- **Text:** Add markdown for instructions or context.
- **Parameters:** Create dropdowns or filters for dynamic queries.
- **Queries:** Use KQL to fetch and visualize data.
- **Visualizations:** Display results as charts, tables, or tiles.

Step 3: Write KQL Queries

KQL is the backbone of your dashboard. It allows you to query logs stored in Azure Data Explorer or Log Analytics. Below are some example queries tailored for security operations.

Example 1: Monitor Failed Login Attempts

This query identifies failed Azure AD sign-in attempts, which could indicate brute-force attacks.

```
SigninLogs
```

```
| where ResultType != 0
```

```
| summarize FailedLogins = count() by UserPrincipalName, AppDisplayName
```

```
| order by FailedLogins desc
```

```
| limit 10
```

- **Explanation:** Filters for failed sign-ins (ResultType != 0), groups by user and application, counts occurrences, and sorts by frequency.
- **Visualization:** Use a **Bar Chart** to show the top 10 users with failed logins.

Example 2: Detect Suspicious Process Execution

This query looks for unusual processes executed on Windows machines.

```
SecurityEvent
```

```
| where EventID == 4688
```

```
| where ProcessName contains "cmd.exe" or  
ProcessName contains "powershell.exe"
```

```
| summarize ProcessCount = count() by Computer,  
ProcessName
```

```
| where ProcessCount > 5
```

- **Explanation:** Filters for process creation events (EventID == 4688), focuses on cmd.exe or powershell.exe, and flags hosts with high execution counts.
- **Visualization:** Use a **Table** to list computers and process counts.

Example 3: Alert Trends Over Time

This query tracks the number of security alerts over the past week.

```
SecurityAlert
```

```
| where TimeGenerated > ago(7d)
```

```
| summarize AlertCount = count() by  
bin(TimeGenerated, 1h), AlertName
```

```
| render timechart
```

- **Explanation:** Groups alerts by hour and name, counts occurrences, and plots them over time.

- **Visualization:** Use a **Time Chart** to show alert trends.

Step 4: Add Queries to Your Workbook

1. In the Workbook editor, click **Add Query**.
2. Paste your KQL query into the query box.
3. Select a **Visualization** (e.g., Bar Chart, Time Chart, Table).
4. Optionally, set **Parameters** (e.g., time range or specific user) to make the query dynamic.
5. Click **Done Editing** to preview the visualization.

Step 5: Customize the Dashboard Layout

- **Add Text:** Use markdown to provide context or instructions (e.g., “This chart shows failed login attempts over the past 24 hours”).
- **Rearrange Elements:** Drag and drop query results, charts, or text blocks to organize the layout.
- **Add Parameters:** Create dropdowns for time ranges, data sources, or specific entities (e.g., users, devices).
 - Example: Add a **Time Range** parameter with options

like “Last 24 hours,” “Last 7 days,” etc.

Step 6: Save and Share

1. Click **Save** and give your Workbook a name (e.g., “SOC Threat Monitoring Dashboard”).
2. Share the Workbook with your team by assigning appropriate permissions in Azure.
3. Optionally, pin the Workbook to your Azure Portal dashboard for quick access.

Example Dashboard: SOC Threat Monitoring

Let’s combine the above queries into a sample SOC dashboard.

Dashboard Components

1. **Header (Text):**
markdown

```
# SOC Threat Monitoring Dashboard
```

Monitor key security metrics, including failed logins, suspicious processes, and alert trends.

- 1. Parameter:** Time Range
 - Options: Last 24 hours, Last 7 days, Last 30 days.
- 2. Visualization 1:** Failed Login Attempts
 - Query: Use the failed login query from Example 1.
 - Visualization: Bar Chart showing top 10 users with failed logins.
- 3. Visualization 2:** Suspicious Process Execution
 - Query: Use the process execution query from Example 2.
 - Visualization: Table listing computers and process counts.
- 4. Visualization 3:** Alert Trends
 - Query: Use the alert trends query from Example 3.
 - Visualization: Time Chart showing alerts over time.

Sample Workbook JSON (Simplified)

If you prefer to import a pre-built Workbook, you can use the Azure Workbook JSON format. Below is a simplified snippet:

{

"version": "Notebook/1.0",

"items": [

{

"type": 1,

"content": {

"json": "# SOC Threat Monitoring
Dashboard\nMonitor key security metrics."

}

} ,

{

"type": 9,

"content": {

"version": "KqlParameterItem/1.0",

"parameters": [

{

"name": "TimeRange",

"label": "Time Range",

"type": 2,

```
    "typeSettings": {  
        "selectableValues": [  
            { "value": "24h", "label":  
                "Last 24 hours" },  
            { "value": "7d", "label": "Last  
7 days" }  
        ]  
    }  
}
```

},

{

"type": 3,

"content": {

"version": "KqlItem/1.0",

"query": "SigninLogs | where ResultType != 0 | summarize FailedLogins = count() by UserPrincipalName, AppDisplayName | order by FailedLogins desc | limit 10",

"size": 1,

"visualization": "barchart"

```
    }
```

```
}
```

```
]
```

```
}
```

- **How to Use:** Copy this JSON, go to the Workbook editor, click **Advanced Editor**, and paste it to import the structure.

Best Practices for Building Effective Dashboards

1. **Keep It Simple:** Focus on key metrics that align with your SOC's priorities (e.g., high-risk alerts, anomalous behavior).
2. **Use Clear Visualizations:** Choose charts that make trends or outliers obvious (e.g., time charts for trends, tables for details).
3. **Leverage Parameters:** Allow analysts to filter data dynamically without editing queries.

4. **Optimize Performance:** Limit query time ranges and use summarization to reduce load times.
5. **Iterate Based on Feedback:** Regularly update the dashboard based on analyst input to ensure it remains relevant.

Enhancing Visibility with Advanced Techniques

To take your dashboards to the next level:

- **Incorporate Machine Learning:** Use Microsoft Sentinel's built-in ML capabilities to detect anomalies and display them in Workbooks.
- **Integrate Threat Intelligence:** Join your KQL queries with threat intelligence feeds to highlight known malicious IPs or domains.
- **Automate Actions:** Link dashboards to Playbooks (Azure Logic Apps) to trigger automated responses for critical alerts.

Custom dashboards built with KQL and Azure Workbooks empower SOC teams to gain deeper visibility into their security environment. By writing targeted KQL queries and leveraging

Workbook visualizations, you can create intuitive, actionable dashboards tailored to your organization's needs. Whether you're monitoring failed logins, tracking suspicious processes, or analyzing alert trends, these tools help you stay one step ahead of threats.

Start small with the examples provided, experiment with different visualizations, and iterate based on your team's feedback. With practice, you'll unlock the full potential of Microsoft Sentinel's analytics and visualization capabilities.

Happy dashboarding!

Chapter 43

Using KQL to Enhance Log Analytics: Best Practices for Filtering, Aggregating, and Visualizing Logs

Organizations generate massive volumes of logs from applications, infrastructure, and other digital systems. Analyzing these logs is crucial for monitoring performance, troubleshooting issues, and gaining insights into user behavior. This is where Kusto Query Language (KQL) comes into play. KQL is a powerful tool designed for querying and analyzing log data stored in various platforms such as Azure Monitor, Application Insights, and Log Analytics.

This chapter will guide you through the best practices for using KQL to filter, aggregate, and visualize logs effectively, ensuring you can harness the full potential of your data.

Best Practices for Filtering Logs

Efficient log analysis begins with filtering. The ability to sift through large datasets to extract relevant information is a cornerstone of effective log analytics. Here are some best practices:

1. Use Filters Early

Filtering early in your query limits the dataset KQL processes, which improves performance. For instance, using the **where** clause at the beginning of your query helps narrow down the dataset right away. Here's an example:

```
AppRequests
```

```
| where Timestamp >= ago(1d) and ResponseCode  
== 500
```

This query filters for application requests within the past day that returned a 500 response code.

2. Be Specific with Conditions

Avoid overly broad filters. Instead, use specific conditions to target precisely what you're looking

for. Combine multiple conditions with `and` or `or` operators to refine your results.

3. Use Time-Based Filters

Logs are often time-series data, so using time constraints like `ago()` or specific date ranges is a critical practice. This ensures that you're analyzing the most relevant data:

```
SysLogs
```

```
| where EventTime between (datetime(2023-01-01)  
.. datetime(2023-01-31))
```

Best Practices for Aggregating Logs

Once logs are filtered, aggregation allows you to summarize data and uncover trends or anomalies. Aggregation is essential for creating meaningful visualizations and reports.

1. Leverage the summarize Operator

The **summarize** operator is a key feature in KQL for aggregation. It allows you to calculate metrics such as averages, counts, and sums. For instance:

SysLogs

```
| summarize TotalErrors = count() by EventType
```

This query counts the number of errors for each type of event.

2. Group Data Effectively

Use grouping wisely to make your aggregations more insightful. Group by dimensions such as time, application, or user. For example:

AppRequests

```
| summarize RequestCount = count() by bin(Timestamp, 1h)
```

Here, the `bin()` function groups data into hourly intervals, making it easier to detect patterns over time.

3. Combine Aggregations

KQL allows combining multiple aggregations in a single query to derive richer insights. For example:

```
AppRequests
```

```
| summarize AvgDuration = avg(Duration),  
TotalRequests = count() by ResponseCode
```

This query calculates the average duration and total count of requests, grouped by response code.

Best Practices for Visualizing Logs

Visualization is where raw data transforms into actionable insights. KQL integrates seamlessly with platforms like Azure Monitor and

Application Insights to create compelling visualizations.

1. Use Time Charts for Time-Series Data

Time-series data is best represented using line charts. By binning data into time intervals and using the **render** operator, you can create visualizations directly in KQL:

```
AppRequests
```

```
| summarize RequestCount = count() by  
bin(Timestamp, 1h)
```

```
| render timechart
```

2. Choose the Right Visualization for Your Data

Different data types require different visualizations. For instance:

- **Pie Charts:** Best for showing proportions or distributions.
- **Bar Charts:** Ideal for comparing categories.
- **Line Charts:** Perfect for trends over time.

Always match the visualization type to your analysis goal.

3. Use Annotations for Context

When visualizing data, add annotations or thresholds to highlight key points or anomalies. This helps provide context to stakeholders reviewing the data.

4. Export and Share Insights

Once your visualizations are ready, use features like dashboards in Azure Monitor to share insights with your team. This ensures that everyone has access to the data they need for decision-making.

Advanced Tips

For advanced KQL users, the following tips can enhance your queries:

1. Use Functions for Reusability

Create reusable functions for common queries. This helps maintain consistency and simplifies complex analyses.

2. Experiment with Machine Learning

KQL supports machine learning capabilities like anomaly detection. For example:

```
AppRequests
```

```
| summarize Count = count() by bin(Timestamp,  
1h)
```

```
| extend Anomaly = iff(Count > 1000, "Yes",  
"No")
```

3. Combine Data Sources

Merge data from multiple datasets using the `join` operator to perform cross-source analysis.

Kusto Query Language is an indispensable tool for log analytics, offering powerful capabilities to filter, aggregate, and visualize data. By following the best practices outlined in this chapter, you can unlock deeper insights from your logs, improve system performance monitoring, and make data-driven decisions with confidence.

Whether you're a seasoned data analyst or a newcomer to log analytics, mastering KQL will undoubtedly elevate your expertise and effectiveness. Start exploring, experimenting, and enhancing your log analytics workflow today!

Chapter 44

Harnessing the KQL let Operator for Testing and Learning

If you're diving into Kusto Query Language (KQL) to analyze data in Azure Data Explorer, Microsoft Sentinel, or other Kusto-powered platforms, the let operator is a powerful tool to streamline your queries and enhance your learning experience. One of its most practical applications is creating synthetic datasets for testing and experimentation. In this chapter, we'll explore how to use the let operator to craft datasets, why it's useful, and provide practical examples to get you started.

Why Use the let Operator for Datasets?

The let operator in KQL allows you to define variables, which can hold scalar values, tabular results, or even functions. When it comes to testing and learning, creating synthetic datasets with let offers several benefits:

- **Controlled Data:** You can design datasets with specific characteristics to test queries without relying on real data, which may be sensitive or incomplete.
- **Reusability:** Store datasets as variables to reuse across multiple queries, saving time and reducing redundancy.
- **Learning Environment:** Practice KQL concepts like joins, aggregations, or time-series analysis with predictable data you create.
- **Debugging:** Test edge cases or specific scenarios to understand how KQL operators behave.

Let's dive into how to use let to create datasets and see it in action.

Creating a Simple Dataset with let

The let operator can be paired with the datatable operator to create a tabular dataset. The datatable operator allows you to define a table with columns and rows directly in your query.

Here's a basic example of creating a dataset to simulate user activity logs:

```
let UserActivity = datatable(UserId: string,  
Activity: string, Timestamp: datetime)
```

```
[
```

```
    "U001", "Login", datetime(2025-06-  
25T08:00:00Z),
```

```
    "U001", "FileAccess", datetime(2025-06-  
25T08:15:00Z),
```

```
    "U002", "Login", datetime(2025-06-  
25T09:00:00Z),
```

```
    "U002", "Logout", datetime(2025-06-  
25T09:30:00Z)
```

```
];
```

```
UserActivity
```

What's happening here?

- We define a variable `UserActivity` using `let`.
- The `datatable` operator creates a table with three columns: `UserId` (string), `Activity` (string), and `Timestamp` (`datetime`).
- We populate the table with four rows of data.
- Finally, we output the `UserActivity` dataset.

This dataset can now be used to practice filtering, grouping, or time-based queries. For example, to count activities per user:

```
let UserActivity = datatable(UserId: string,  
Activity: string, Timestamp: datetime)
```

```
[
```

```
"U001", "Login", datetime(2025-06-  
25T08:00:00Z),
```

```
"U001", "FileAccess", datetime(2025-06-  
25T08:15:00Z),  
  
"U002", "Login", datetime(2025-06-  
25T09:00:00Z),  
  
"U002", "Logout", datetime(2025-06-  
25T09:30:00Z)  
];
```

UserActivity

```
| summarize ActivityCount = count() by UserId
```

Output:

UserId	ActivityCount
U001	2
U002	2

Generating Larger Datasets with range

For more complex testing, you might need a larger dataset. KQL's range operator, combined with let and make-series, can help generate synthetic data programmatically. Here's an example of creating a dataset with simulated temperature readings:

```
let TemperatureData = range Timestamp from  
ago(7d) to now() step 1h  
  
| extend SensorId = "S001",  
  
    Temperature = rand() * 20 + 15; //  
Random temperature between 15°C and 35°C  
  
TemperatureData  
  
| summarize AvgTemperature = avg(Temperature)  
by SensorId, bin(Timestamp, 1d)
```

What's happening here?

- The range operator generates a series of timestamps from 7 days ago to now, with a 1-hour step.
- We use extend to add a SensorId column and a Temperature column with random values between 15°C and 35°C.
- The dataset is stored in the TemperatureData variable.
- We then summarize the average temperature per sensor per day.

This approach is ideal for testing time-series analysis, aggregations, or anomaly detection without needing real sensor data.

Using let for Reusable Test Scenarios

The let operator shines when you want to create reusable datasets for multiple test scenarios. For example, suppose you're learning about joins. You can create two datasets and experiment with different join types:

```
let Orders = datatable(OrderId: int,  
CustomerId: string, Amount: real)
```

```
[  
    1, "C001", 100.50,  
    2, "C002", 200.75,  
    3, "C001", 150.25  
];  
  
let Customers = datatable(CustomerId: string,  
Name: string)  
[  
    "C001", "Alice",  
    "C003", "Bob"
```

```
];
```

Orders

```
| join kind=leftouter Customers on CustomerId
```

Output:

OrderId	CustomerId	Amount	CustomerId1	Name
1	C001	100.50	C001	Alice
2	C002	200.75		
3	C001	150.25	C001	Alice

Here, the Orders and Customers datasets are stored as variables, making it easy to test inner, leftouter, or other join types without redefining the data.

Tips for Using let in Testing and Learning

1. **Start Small:** Begin with simple datasets to understand KQL operators, then scale up to more complex scenarios.

-
- 2. Simulate Real-World Data:** Mimic the structure of your actual data (e.g., timestamps, IDs, metrics) to make your tests relevant.
- 3. Combine with Functions:** Use let to define reusable functions that generate or transform datasets dynamically.
- 4. Document Your Datasets:** Add comments to your queries to describe the dataset's purpose and structure for future reference.
- 5. Experiment Freely:** Since synthetic data isn't real, you can test extreme cases or errors without consequences.

The KQL let operator, combined with tools like datatable and range, is a game-changer for creating synthetic datasets for testing and learning. Whether you're practicing aggregations, joins, or time-series analysis, let enables you to craft controlled, reusable datasets that accelerate your mastery of KQL. By experimenting with these techniques, you'll gain confidence in writing queries and tackling real-world data challenges.

Try creating your own datasets with let and share your favorite use cases! Happy querying!

Chapter 45

Automating Alerts with KQL in Azure Monitor: A Step-by-Step Guide

Monitoring application performance and infrastructure health is critical for maintaining reliable systems. Azure Monitor provides a powerful platform to collect, analyze, and act on telemetry data, and its Kusto Query Language (KQL) enables precise, customizable queries to detect issues. In this chapter, I'll walk you through setting up automated alerts in Azure Monitor using KQL to track application performance or infrastructure issues, ensuring you can respond proactively to potential problems.

Why Use KQL for Alerts in Azure Monitor?

Azure Monitor aggregates metrics, logs, and traces from your Azure resources, applications, and infrastructure. KQL, the query language used in Azure Data Explorer and Azure Monitor, allows you to write sophisticated queries to filter and

analyze this data. By combining KQL with Azure Monitor's alerting capabilities, you can:

- **Detect anomalies** in application performance (e.g., high error rates, slow response times).
- **Monitor infrastructure health** (e.g., CPU spikes, memory exhaustion).
- **Automate notifications** to teams via email, SMS, or integrations like Microsoft Teams or PagerDuty.
- **Reduce manual monitoring** by triggering alerts based on specific conditions.

Let's dive into the process of setting up an alert using KQL in Azure Monitor.

Prerequisites

Before you begin, ensure you have:

- An **Azure subscription** with access to Azure Monitor.
- A resource (e.g., an Azure App Service, VM, or Kubernetes cluster) generating telemetry data.
- **Log Analytics workspace** configured to store logs and metrics.
- Basic familiarity with KQL (don't worry, I'll provide example queries).

- Permissions to create alerts in Azure Monitor.

Step 1: Understand Your Monitoring Needs

First, identify what you want to monitor.

Common scenarios include:

- **Application performance:**
 - High error rates in an API (e.g., HTTP 500 errors).
 - Slow response times for user requests.
 - Increased latency in a web app.
- **Infrastructure issues:**
 - CPU usage exceeding 80% for a virtual machine.
 - Disk space running low on a server.
 - Memory leaks in a containerized application.

For this walkthrough, let's assume we're monitoring an **Azure App Service** and want to trigger an alert if the **average response time exceeds 2 seconds over a 5-minute period**.

Step 2: Write a KQL Query

KQL queries are the foundation of your alert. You'll write a query to analyze telemetry data in your Log Analytics workspace and return results when the condition is met.

Example Scenario

We want to monitor the requests table in Azure Monitor Logs, which captures HTTP request data for our App Service. The goal is to detect when the average response time (duration) exceeds 2000 milliseconds (2 seconds) over a 5-minute window.

Here's a KQL query to achieve this:

```
requests
```

```
| where timestamp > ago(5m)
```

```
| summarize AvgResponseTime = avg(duration) by bin(timestamp, 5m)
```

```
| where AvgResponseTime > 2000
```

Query Breakdown

- requests: The table containing HTTP request data.
- where timestamp > ago(5m): Filters data from the last 5 minutes.
- summarize AvgResponseTime = avg(duration) by bin(timestamp, 5m): Calculates the average response time in 5-minute intervals.
- where AvgResponseTime > 2000: Returns results only when the average response time exceeds 2000ms.

Testing the Query

1. Go to the **Azure Portal > Log Analytics workspace > Logs**.
2. Paste the KQL query and run it.
3. Verify that the query returns results only when the condition is met (e.g., response time > 2 seconds). If no results are returned, the condition isn't currently met, which is expected in a healthy system.

Step 3: Create an Alert Rule

Now that you have a working KQL query, let's create an alert rule in Azure Monitor.

Navigate to Azure Monitor

1. In the **Azure Portal**, go to **Monitor > Alerts > Create > Alert rule**.
2. Select the scope (e.g., your App Service or Log Analytics workspace) and click **Next**.

Define the Condition

1. Under **Condition**, click **Add condition**.
2. Choose **Log search alert** as the signal type.
3. In the **Search query** field, paste your KQL query:

requests

```
| where timestamp > ago(5m)
```

```
| summarize AvgResponseTime = avg(duration) by  
bin(timestamp, 5m)
```

```
| where AvgResponseTime > 2000
```

4. Configure the **Measure**:

- Set **Aggregation type** to Count of rows.
 - Set **Aggregation granularity** to 5 minutes (to match the query's time window).
5. Set the **Threshold**:
- **Operator:** Greater than.
 - **Threshold value:** 0 (since any returned rows indicate the condition is met).
 - Set the **Evaluation frequency**:
 - **Check every:** 5 minutes.
 - **Look-back period:** 5 minutes (to align with the query's time range).

Configure Actions

1. Under **Actions**, click **Add action groups**.
2. Create a new action group or select an existing one.
3. Define the action:
 - **Email/SMS:** Send an email or SMS to your team.
 - **Webhook:** Trigger a webhook for tools like Microsoft Teams or PagerDuty.
 - **Azure Function:** Run automated remediation (e.g., scale up resources).
4. Save the action group.

Set Alert Details

1. Provide an **Alert rule name** (e.g., HighResponseTimeAlert).
2. Add a **Description** (e.g., "Alerts when App Service response time exceeds 2 seconds").
3. Select the **Severity** (e.g., Sev 2 for moderate issues).
4. Choose the **Resource group** and **Region** for the alert rule.
5. Enable the alert rule upon creation.

Review and Create

Review the configuration and click **Create**. Your alert rule is now active!

Step 4: Test the Alert

To ensure the alert works, simulate the condition (if possible) or monitor it in a real-world scenario. For example:

- Introduce artificial latency in your App Service (e.g., in a test environment).
- Check the **Alerts** tab in Azure Monitor to confirm the alert fires when response time exceeds 2 seconds.
- Verify that notifications are sent to the configured action group (e.g., email or Teams).

If the alert doesn't fire, revisit your KQL query or alert rule settings to ensure the threshold and evaluation frequency are correct.

Step 5: Refine and Expand

Once your alert is working, consider refining it or adding more alerts for other scenarios. Here are some additional KQL query examples:

Monitor HTTP 500 Errors

Alert if the number of HTTP 500 errors exceeds 10 in a 5-minute period:

```
requests
```

```
| where timestamp > ago(5m)
```

```
| where resultCode == "500"
```

```
| summarize ErrorCount = count() by  
bin(timestamp, 5m)
```

```
| where ErrorCount > 10
```

Monitor VM CPU Usage

Alert if CPU usage on a virtual machine exceeds 80% for 5 minutes:

Perf

```
| where ObjectName == "Processor" and  
CounterName == "% Processor Time"
```

```
| where timestamp > ago(5m)
```

```
| summarize AvgCPU = avg(CounterValue) by  
bin(timestamp, 5m), Computer
```

```
| where AvgCPU > 80
```

Monitor Disk Space

Alert if free disk space on a server falls below 10%:

```
InsightsMetrics
```

```
| where Name == "FreeSpacePercentage"
```

```
| where timestamp > ago(5m)
```

```
| summarize AvgFreeSpace = avg(Val) by  
bin(timestamp, 5m), Computer
```

```
| where AvgFreeSpace < 10
```

Best Practices for KQL Alerts

1. **Optimize Queries:** Ensure KQL queries are efficient to avoid performance issues in large datasets. Use filters like `timestamp > ago()` early in the query.
2. **Set Appropriate Thresholds:** Avoid false positives by testing thresholds in your environment.
3. **Use Action Groups:** Centralize notification logic by reusing action groups across multiple alerts.

4. **Leverage Severity Levels:** Assign appropriate severity (e.g., Sev 0 for critical, Sev 4 for informational) to prioritize responses.
5. **Monitor Alert Health:** Periodically review fired alerts to ensure they're still relevant and not generating noise.
6. **Integrate with Automation:** Use Azure Logic Apps or Azure Functions to automate responses, such as restarting a service or scaling resources.

Troubleshooting Common Issues

- **No data returned by query:** Ensure your resource is sending telemetry to the Log Analytics workspace and the table (e.g., requests) is populated.
- **Alerts not firing:** Check the threshold, evaluation frequency, and look-back period. Verify the query returns results when the condition is met.
- **Too many alerts:** Adjust the threshold or aggregation granularity to reduce noise.
- **Notifications not received:** Confirm the action group is correctly configured and test the notification channel.

Automating alerts with KQL in Azure Monitor empowers you to proactively monitor application performance and infrastructure health. By writing targeted KQL queries and configuring alert rules, you can detect issues early and notify the right teams—or even trigger automated remediation. Whether you’re tracking slow response times, server errors, or resource exhaustion, Azure Monitor’s flexibility makes it a powerful tool for maintaining system reliability.

Start small with a single alert, test it thoroughly, and expand to cover more scenarios as you gain confidence. With KQL and Azure Monitor, you’re well-equipped to keep your systems running smoothly.

Chapter 46

KQL Query Optimization: Techniques to Boost Efficiency and Slash Processing Time

Kusto Query Language (KQL) is a powerful tool for querying and analyzing large datasets in Azure Data Explorer, Synapse Data Explorer, and other Microsoft services. However, as datasets grow and queries become more complex, inefficient KQL queries can lead to slow performance, increased resource consumption, and higher costs. Optimizing your KQL queries is essential to ensure fast processing and efficient resource utilization. In this chapter, we'll explore practical techniques to make your KQL queries more efficient and reduce processing time.

Why Optimize KQL Queries?

Optimizing KQL queries offers several benefits:

- **Faster Results:** Reduced query execution time means quicker insights.
- **Cost Savings:** Efficient queries consume fewer resources, lowering compute costs in cloud environments.

- **Scalability:** Optimized queries handle larger datasets and higher query volumes effectively.
- **Better User Experience:** Faster response times improve usability for end-users and analysts.

Let's dive into the top techniques for optimizing KQL queries.

1. Use Filters Early with where Clauses

Filtering data as early as possible in your query reduces the amount of data processed in subsequent steps. The where clause is your first line of defense for eliminating irrelevant rows.

Tips:

- Place where clauses immediately after the data source to minimize the dataset.
- Use specific conditions (e.g., exact matches or ranges) to narrow down results.
- Leverage time-based filters for time-series data, as KQL is optimized for such operations.

Example:

Instead of:

```
StormEvents
```

```
| summarize count() by State
```

```
| where count_ > 100
```

Filter first:

```
StormEvents
```

```
| where EventType == "Tornado"
```

```
| summarize count() by State
```

```
| where count_ > 100
```

**This reduces the dataset before the costly
summarize operation.**

2. Limit Columns with project or project-away

KQL processes all columns in a table unless explicitly limited. Use project to select only the columns you need or project-away to exclude unnecessary ones. This reduces memory usage and speeds up query execution.

Tips:

- Use project early in the query to reduce the number of columns passed to downstream operators.
- Avoid selecting all columns (*) in production queries.

Example:

Instead of:

StormEvents

```
| where EventType == "Flood"
```

```
| summarize count() by State
```

Select specific columns:

```
StormEvents
```

```
| where EventType == "Flood"
```

```
| project State
```

```
| summarize count() by State
```

This avoids processing unused columns like EventId or BeginDateTime.

3. Optimize Joins with hint and Appropriate Join Types

Joins can be resource-intensive, especially with large datasets. Using the right join type and providing hints can significantly improve performance.

Tips:

- Use inner joins instead of outer joins when possible, as they process fewer rows.
- Apply filters before joining to reduce the dataset.
- Use hint.strategy to guide the engine, such as hint.strategy=shuffle for large datasets or hint.strategy=broadcast for small lookup tables.

Example:

Instead of:

```
StormEvents
```

```
| join Locations on State
```

```
| where EventType == "Hurricane"
```

Filter and optimize:

```
StormEvents
```

```
| where EventType == "Hurricane"  
  
| project State, EventId  
  
| join hint.strategy=shuffle (Locations |  
project State, LocationId) on State
```

This filters data early and uses a shuffle strategy for better performance.

4. Leverage Materialized Views for Frequent Queries

Materialized views precompute and store results for frequently used queries, reducing processing time for repetitive operations.

Tips:

- Create materialized views for common aggregations or filters.
- Ensure the view is updated with a suitable refresh policy to balance freshness and performance.

Example:

Create a materialized view for tornado events by state:

```
.create materialized-view TornadoSummary on  
table StormEvents
```

```
{
```

```
StormEvents
```

```
| where EventType == "Tornado"
```

```
| summarize EventCount=count() by State
```

```
}
```

Query the view directly:

```
TornadoSummary
```

```
| where EventCount > 50
```

This avoids recomputing the aggregation each time.

5. Use summarize Efficiently

The summarize operator is powerful but can be costly if not used wisely. Optimize aggregations to minimize computation.

Tips:

- Reduce the dataset with where or project before summarizing.
- Use approximate aggregations (e.g., dcount with hll) for large datasets when exact counts aren't required.
- Avoid grouping by high-cardinality columns unless necessary.

Example:

Instead of:

```
StormEvents
```

```
| summarize count() by EventId
```

Use:

```
StormEvents
```

```
| where EventType == "Thunderstorm"  
  
| summarize approx_count=dcount(EventId, 0.01)
```

This uses an approximate count to reduce processing time.

6. Partition Data for Time-Series Queries

KQL excels at time-series analysis, and partitioning data by time can drastically improve performance.

Tips:

- Use datetime filters to leverage partitioning (e.g., where Timestamp > ago(7d)).
- Ensure tables are partitioned by time when ingesting data.

- Use restrict clauses for queries spanning specific time ranges.

Example:

Instead of:

```
StormEvents
```

```
| where EventType == "Flood"
```

Use a time filter:

```
StormEvents
```

```
| where StartTime > ago(30d)
```

```
| where EventType == "Flood"
```

This leverages time-based partitioning for faster execution.

7. Avoid Unnecessary Sorting with sort or order

Sorting is computationally expensive, especially on large datasets. Only use sort or order when necessary.

Tips:

- Use top instead of sort if you only need a few rows.
- Limit the dataset before sorting.
- Avoid sorting by high-cardinality or complex expressions.

Example:

Instead of:

```
StormEvents
```

```
| sort by DamageProperty desc
```

```
| take 10
```

Use:

```
StormEvents
```

```
| top 10 by DamageProperty desc
```

This is more efficient as it avoids sorting the entire dataset.

8. Cache Results for Repeated Queries

If a query is run frequently with the same parameters, enable caching to store results and avoid recomputation.

Tips:

- Use the set statement with query_caching to enable caching.
- Cache results for static or slowly changing datasets.

Example:

```
set query_caching = "enabled";
```

```
StormEvents
```

```
| where EventType == "Hurricane"  
  
| summarize count() by State
```

This caches the result for subsequent executions.

9. Monitor and Analyze Query Performance

Use KQL's built-in tools to diagnose and optimize slow queries.

Tips:

- Use the .show queries command to review query execution details.
- Analyze the Query Execution Plan to identify bottlenecks (e.g., excessive shuffling or scanning).
- Enable query diagnostics with | render diagnostics to visualize performance metrics.

Example:

```
StormEvents
```

```
| where EventType == "Tornado"  
  
| summarize count() by State  
  
| render diagnostics
```

This provides insights into query execution stages and resource usage.

10. Test and Iterate with Small Datasets

Before running queries on large datasets, test them on smaller subsets to ensure correctness and performance.

Tips:

- Use `take` or `sample` to work with a subset of data.
- Validate query logic before scaling to full datasets.

- Measure execution time with .show query performance.

Example:

```
StormEvents
```

```
| where EventType == "Flood"
```

```
| take 1000
```

```
| summarize count() by State
```

This tests the query on a small sample before running it on the full dataset.

Optimizing KQL queries is both an art and a science. By applying techniques like early filtering, column pruning, efficient joins, and leveraging features like materialized views and caching, you can significantly reduce processing time and resource consumption. Always monitor

query performance and test iteratively to ensure your queries remain efficient as datasets grow.

Start applying these techniques today to unlock the full potential of KQL and make your data analysis faster, cheaper, and more scalable.

Happy querying!

Chapter 47

Using KQL to Query Large Datasets Efficiently

In the age of big data, efficient querying of vast datasets is critical for unlocking insights and driving decision-making. Kusto Query Language (KQL), developed for Azure Data Explorer and Log Analytics, is a powerful tool designed for querying large datasets with speed and precision. However, as the size of datasets grows, performance bottlenecks can arise, impacting efficiency and usability. This chapter explores strategies and best practices for using KQL to query large datasets effectively while addressing performance challenges.

Understanding KQL and Its Strengths

KQL excels in scenarios requiring complex filtering, aggregation, and visualization of data. It is particularly adept at handling log and

telemetry data, making it a popular choice for monitoring and analytics applications. Its intuitive syntax, akin to SQL, and its highly optimized execution engine contribute to its ability to manage sizable datasets. However, leveraging KQL's full potential requires thoughtful query design and an understanding of its limitations.

Challenges of Querying Large Datasets

Large datasets bring unique challenges:

- **High computational cost:** Processing billions of rows can strain resources, leading to slower query execution.
- **Memory limitations:** Operations requiring extensive memory, such as sorting or joining large tables, can cause resource bottlenecks.
- **Network latency:** Retrieving large result sets across networks can degrade performance.

To mitigate these challenges, adopting efficient querying strategies is essential.

Strategies for Efficient KQL Querying

1. Start with Selective Data Filtering

One of the most effective ways to optimize KQL queries is to reduce the volume of data processed. Use selective filtering to narrow down the dataset upfront:

- **Leverage time filters:** Always filter by time range using the where clause. For example:

```
| where Timestamp between (datetime(2025-01-01)  
.. datetime(2025-01-31))
```

- **Apply specific conditions:** Add precise conditions to focus on relevant data. For instance:

```
| where EventType == "Error" and UserId ==  
"12345"
```

2. Use Summarization and Aggregation Early

Instead of returning raw data, summarize and aggregate it as early as possible in your query:

- Use the summarize operator to calculate metrics such as counts, averages, or sums:

```
| summarize ErrorCode = count() by EventType
```

- Minimize unnecessary row expansion by aggregating data before performing joins or additional transformations.

3. Optimize Joins

Joins can significantly impact query performance. To optimize them:

- Filter each table individually before joining.
- Use the lookup operator when possible, as it is more efficient for small-to-large joins:

```
| lookup kind=leftouter (OtherTable) on  
KeyColumn
```

4. Limit Result Sets

When querying large datasets, always limit the number of rows returned:

- Use the take operator to cap results for testing:

```
| take 1000
```

- For production scenarios, use top with sorting to retrieve the most relevant results:

```
| top 100 by Timestamp desc
```

5. Partition Data for Parallel Processing

Partitioning data enables parallel processing, reducing query execution time:

- Use the extend operator to create partitions:

```
| extend PartitionKey = hash(UserId, 10)
```

- Distribute operations across partitions using the summarize or union operators.

6. Profile and Tune Queries

Profiling helps identify bottlenecks and optimize performance:

- Use the explain or set query_statistics commands to analyze query execution plans.
- Adjust query structure based on insights from profiling, such as reordering filters or redefining joins.

Advanced Techniques

1. Materialized Views

Materialized views store pre-computed results, enabling faster query execution for frequently accessed data. Configure a materialized view for high-cost calculations or recurring queries.

2. Leverage Caching

KQL supports result caching for queries run repeatedly. Utilize caching to minimize processing overhead for repeated analyses or dashboards.

3. Use Functions for Modularity

Define reusable functions for complex, multi-step queries. This improves readability and performance through modular design:

```
.create function ErrorSummary() {  
    Logs  
    | where EventType == "Error"  
    | summarize ErrorCode = count() by UserId  
}
```

Invoke the function in subsequent queries:

```
ErrorSummary()
```

```
| where ErrorCount > 10
```

4. Optimize Data Ingestion

Efficient data ingestion impacts downstream query performance. Pre-process data to remove unnecessary fields, apply compression, and ensure schema consistency before ingestion.

Common Pitfalls to Avoid

- **Unfiltered full scans:** Avoid running queries without filters, as they process entire datasets unnecessarily.
- **Over-reliance on client-side tools:** Perform data transformations in KQL rather than exporting large datasets for external processing.
- **Excessive joins:** Minimize joins, especially those involving multiple large tables.

KQL is a robust tool for querying large datasets, but efficient use requires strategic query design and optimization. By incorporating selective filtering, early summarization, optimized joins, and advanced techniques like materialized views

and caching, you can overcome performance bottlenecks and unlock the full potential of your data. In big data scenarios, a thoughtful approach to KQL not only improves query performance but also enhances the usability and scalability of your analytics workflows.

Start implementing these strategies today to tackle your big data challenges with confidence and efficiency!

Chapter 48

Understanding KQL Datatypes & Operators

KQL, or Kusto Query Language, is the backbone of querying in Azure Data Explorer and other platforms that rely on structured data exploration. It offers a rich set of features for data manipulation, analysis, and visualization. At the heart of KQL's capabilities are datatypes and operators, which allow users to extract valuable insights from datasets efficiently. This chapter delves deep into KQL datatypes, casting and conversion techniques, and operator usage to help you master the art of crafting precise and powerful queries.

The Fundamentals of KQL Datatypes

Overview of Datatypes

KQL supports a variety of datatypes designed for handling different kinds of data. By

understanding these datatypes, you can structure your queries to leverage the full potential of KQL. The primary datatypes include:

- **String:** Textual data, such as names, titles, or descriptions.
- **Int:** Integer values for whole numbers.
- **Long:** Larger integer values for extended numerical ranges.
- **Real:** Floating-point numbers for precision calculations.
- **Datetime:** Date and time values for temporal data.
- **Timespan:** Durations between events or moments.
- **Boolean:** True or False values for logical operations.
- **Dynamic:** JSON-like data structures for semi-structured data.

Why Datatypes Matter

Understanding datatypes is crucial because mismatched types can lead to errors or inefficient queries. For example, attempting to compare a string to an integer or performing mathematical operations on a datetime value will result in unexpected outcomes. Proper datatype

management ensures data accuracy and query performance.

Casting and Conversion in KQL

The Need for Casting and Conversion

In real-world datasets, data often comes in formats that are not immediately compatible with the intended analysis. Casting and conversion allow you to transform data into appropriate types, enabling seamless operations. For instance, converting a string representation of a date into a datetime datatype opens up possibilities for chronological analysis.

Casting Functions

KQL provides several functions for datatype casting:

- **toint()**: Converts a string into an integer.
- **toreal()**: Converts a string into a floating-point number.
- **todatetime()**: Converts a string into a datetime value.

- **tostring()**: Converts any datatype into a string.
- **totimespan()**: Converts a string or numeric value into a timespan.

Type Conversion Best Practices

When performing type conversions, consider these best practices:

- **Validate Input**: Always ensure the input data matches the expected format.
- **Handle Null Values**: Anticipate cases where the conversion might return null, especially when the data is inconsistent.
- **Use Explicit Casting**: Avoid implicit conversions, as they can lead to ambiguity and errors in complex queries.

Exploring KQL Operators

Types of Operators

Operators in KQL define relationships between data points or execute calculations. They can be categorized as follows:

- **Arithmetic Operators**: +, -, *, / for mathematical operations.

- **Comparison Operators:** ==, !=, , >= for evaluating conditions.
- **Logical Operators:** and, or, not for combining or negating conditions.
- **String Operators:** contains, startswith, endswith for text analysis.
- **Datetime Operators:** datetime_diff(), datetime_add() for manipulating temporal values.

Operator Precedence

Operator precedence determines the order of evaluation in a query. For example, in the expression **x + y * z**, multiplication takes precedence over addition. Use parentheses to explicitly control precedence, ensuring correct calculations: **(x + y) * z**.

Combining Operators

KQL allows chaining multiple operators within a single query. For example:

```
datatable(name:string, age:int, income:real)
```

```
"Alice", 30, 50000.5,  
      "Bob", 25, 45000.0,  
      "Charlie", 35, 60000.3  
    ]  
  
| where age > 25 and income > 48000.0  
  
| project name, age, income
```

This query filters rows based on age and income criteria while projecting selected columns.

Practical Examples

Case Study: Data Cleaning

Suppose you have a dataset where dates are stored as strings. To perform chronological

analysis, you must convert these strings into datetime values. Using `todatetime()`, you can transform the data and apply operators such as `datetime_diff()` for insights into time intervals.

Case Study: Text Analysis

Imagine analyzing customer feedback stored as strings. You can use operators like `contains` or `startswith` to filter specific keywords, enabling sentiment analysis.

Common Pitfalls and How to Avoid Them

Datatype Mismatches

One common error involves comparing incompatible datatypes. For instance, attempting to apply arithmetic operators to datetime values will result in errors. Always ensure that the types involved are compatible.

Handling Null Values

Null values can propagate through operators, leading to unexpected results. Use functions

like `isnotempty()` or `coalesce()` to manage nulls effectively.

Performance Considerations

Complex conversions and operator chains can impact query performance. Optimize queries by minimizing conversions and reducing redundant operators.

Mastering KQL datatypes and operators is an essential skill for efficient and accurate data exploration. By understanding the nuances of casting, conversion, and operator usage, you can craft queries that unlock the full potential of your datasets. Whether you're cleaning data, analyzing trends, or extracting insights, KQL's robust framework empowers you to achieve your goals with precision.

Chapter 49

Correlating Logs Across Azure Services with KQL

Applications often span multiple Azure services, generating logs in various locations such as Azure Application Insights, Log Analytics, and Blob Storage. Correlating these logs is critical for gaining a unified view of system behavior, troubleshooting issues, and identifying performance bottlenecks. Azure's Kusto Query Language (KQL) provides a powerful way to join and analyze logs across these services. In this chapter, we'll explore how to use KQL to correlate logs from Azure Application Insights, Log Analytics, and Blob Storage, enabling comprehensive insights into your Azure environment.

Why Correlate Logs Across Azure Services?

Logs from different Azure services capture distinct aspects of an application:

- **Azure Application Insights:** Tracks application performance, telemetry, and user interactions, such as requests, exceptions, and traces.
- **Log Analytics:** Collects infrastructure and diagnostic logs from Azure resources, including VMs, containers, and Azure Monitor.
- **Blob Storage:** Stores raw or archival logs, such as audit logs, application logs, or custom data exports.

Correlating these logs helps you:

- Trace an issue from a user request (Application Insights) to infrastructure events (Log Analytics) and archived data (Blob Storage).
- Identify patterns or anomalies across services.
- Build end-to-end monitoring dashboards for holistic observability.

KQL, used by Azure Monitor and Azure Data Explorer, is ideal for querying and joining these disparate log sources due to its flexibility and performance.

Prerequisites

Before diving into KQL queries, ensure you have:

- 1. Access to Logs:**
 - Application Insights instance with telemetry data.
 - Log Analytics workspace with resource logs.
 - Blob Storage account with logs exported (e.g., via Diagnostic Settings or custom exports).
- 2. Azure Data Explorer or Log Analytics Workspace:** To run KQL queries.
- 3. Permissions:** Read access to Application Insights, Log Analytics, and Blob Storage.
- 4. External Data Setup:** For Blob Storage, logs must be accessible via Azure Data Explorer or ingested into a Log Analytics workspace.

Step 1: Understanding Log Sources and Schemas

Each Azure service stores logs in a specific format, and KQL queries rely on understanding their schemas:

- **Application Insights:** Logs are stored in tables like requests, traces, exceptions, and customEvents. Key columns include

timestamp, operation_Id, and correlation_Id.

- **Log Analytics:** Logs are stored in tables like AzureDiagnostics, Heartbeat, or custom tables. Common columns include TimeGenerated, ResourceId, and OperationName.
- **Blob Storage:** Logs are typically stored as JSON, CSV, or text files. To query them with KQL, you need to ingest them into Azure Data Explorer or map them as external tables.

To explore schemas, run simple KQL queries in the Log Analytics or Azure Data Explorer query editor:

```
// Application Insights: View recent requests  
  
requests | take 10
```

```
// Log Analytics: View Azure Diagnostics
```

```
AzureDiagnostics | take 10
```

For Blob Storage logs, you'll need to set up an external table or ingest the data. We'll cover this in Step 3.

Step 2: Joining Logs from Application Insights and Log Analytics

To correlate logs between Application Insights and Log Analytics, you can use common fields like operation_Id, correlation_Id, ResourceId, or timestamp. For example, let's say you want to correlate failed requests in Application Insights with diagnostic logs in Log Analytics to investigate an issue.

Here's a sample KQL query to join the two:

```
// Join Application Insights requests with Log Analytics diagnostics
```

```
let FailedRequests = requests
```

```
| where success == false
```

```
| project timestamp, operation_Id, url,  
client_IP;
```

FailedRequests

```
| join kind=inner (
```

AzureDiagnostics

```
| where OperationName == "Request"
```

```
| project TimeGenerated, ResourceId,  
OperationName, CorrelationId
```

```
) on $left.operation_Id == $right.CorrelationId
```

```
| project timestamp, url, client_IP,  
ResourceId, OperationName  
  
| order by timestamp desc
```

Explanation:

- `requests`: Filters failed requests from Application Insights.
- `AzureDiagnostics`: Queries diagnostic logs from Log Analytics.
- `join kind=inner`: Matches records where `operation_Id` (from Application Insights) equals `CorrelationId` (from Log Analytics).
- `project`: Selects relevant columns for analysis.
- `order by`: Sorts results by timestamp for clarity.

This query helps you trace a failed request to the underlying resource or operation logged in Log Analytics.

Step 3: Querying Blob Storage Logs with KQL

Blob Storage logs are not natively queryable with KQL, so you need to either:

1. **Ingest Logs into Azure Data Explorer:** Use tools like Azure Data Factory or Event Hubs to ingest Blob Storage logs into a table.
2. **Query as External Data:** Map Blob Storage files as an external table in Azure Data Explorer.

Here's how to set up an external table for JSON logs in Blob Storage:

1. **Create an External Table:** In Azure Data Explorer, define an external table that points to your Blob Storage files:

```
.create external table BlobLogs (
```

```
    Timestamp: datetime,
```

```
    LogLevel: string,
```

```
    Message: string,
```

```
CorrelationId: string  
  
)  
  
kind=blob  
  
dataformat=json  
  
(  
  
    h@'https://<storage-  
account>.blob.core.windows.net/<container>;<sas  
-token>'  
  
)
```

2. **Query the External Table:** Once the external table is set up, you can query it like any other KQL table:

```
BlobLogs
```

```
| where LogLevel == "Error"  
  
| project Timestamp, Message, CorrelationId  
  
| take 10
```

3. Correlate with Application Insights and Log Analytics:

Join the Blob Storage logs with Application Insights and Log Analytics using a common field like CorrelationId:

```
let AppErrors = requests  
  
| where success == false  
  
| project timestamp, operation_Id, url;  
  
let BlobErrors = BlobLogs  
  
| where LogLevel == "Error"
```

```
| project Timestamp, Message, CorrelationId;
```

AppErrors

```
| join kind=inner BlobErrors on  
$left.operation_Id == $right.CorrelationId
```

```
| join kind=inner (
```

AzureDiagnostics

```
| project TimeGenerated, ResourceId,  
CorrelationId
```

```
) on $left.operation_Id == $right.CorrelationId
```

```
| project timestamp, url, Message, ResourceId
```

```
| order by timestamp desc
```

Explanation:

- AppErrors: Filters failed requests from Application Insights.
- BlobErrors: Filters error logs from Blob Storage.
- join: Combines the datasets using operation_Id and CorrelationId.
- The final result correlates errors across all three services, providing a unified view.

Step 4: Best Practices for Log Correlation

To ensure effective log correlation with KQL:

1. **Use Consistent Correlation IDs:** Ensure your application and services propagate a common correlation_Id or operation_Id across logs. This is critical for joining datasets.
2. **Standardize Timestamps:** Align timestamp formats across services (e.g., UTC) to avoid mismatches during joins.
3. **Optimize Query Performance:**
 - Use where clauses early to filter data.

- Limit the time range with where timestamp > ago(1h).
 - Avoid excessive joins on large datasets.
4. **Automate Ingestion:** For Blob Storage, automate log ingestion into Azure Data Explorer or Log Analytics using Azure Data Factory or Event Grid.
 5. **Visualize Results:** Export KQL query results to Azure Dashboards or Power BI for real-time monitoring.

Step 5: Example Use Case

Suppose you're troubleshooting a spike in failed API requests. You can:

1. Query Application Insights to identify failed requests:

requests

```
| where success == false
```

```
| summarize Count = count() by url,  
bin(timestamp, 5m)
```

```
| order by timestamp desc
```

2. Correlate with Log Analytics to check for resource issues:

AzureDiagnostics

```
| where ResourceType == "VIRTUALMACHINES" and  
ResultType == "Failed"
```

```
| project TimeGenerated, ResourceId,  
CorrelationId
```

3. Check Blob Storage logs for application-specific errors:

BlobLogs

```
| where LogLevel == "Error" and Message  
contains "API"
```

```
| project Timestamp, Message, CorrelationId
```

4. Join the datasets to trace the issue end-to-end:

```
let FailedRequests = requests  
  
| where success == false  
  
| project timestamp, operation_Id, url;  
  
FailedRequests  
  
| join kind=inner (  
  
    BlobLogs  
  
    | where LogLevel == "Error"  
  
    | project Timestamp, Message, CorrelationId
```

```
) on $left.operation_Id == $right.CorrelationId  
  
| join kind=inner (  
  
|   | where OperationType == "PUT"  
  
|   | where ResultType == "Failed"  
  
| project TimeGenerated, ResourceId,  
CorrelationId  
  
) on $left.operation_Id == $right.CorrelationId  
  
| project timestamp, url, Message, ResourceId
```

This query might reveal that failed requests correlate with VM failures and specific error messages in Blob Storage, pointing to a resource contention issue.

Using KQL to correlate logs across Azure Application Insights, Log Analytics, and Blob Storage empowers you to gain deep insights into your application's behavior. By leveraging common fields like correlation_Id and carefully structuring your queries, you can trace issues across services, identify root causes, and improve system reliability. Whether you're troubleshooting incidents or building proactive monitoring, KQL's flexibility makes it an essential tool for Azure log analysis.

Chapter 50

Common Mistakes in KQL & How to Avoid Them: Debugging Strategies for Typical Pitfalls

Kusto Query Language (KQL) has become indispensable for querying data in log analytics, Azure Monitor, and other platforms. While KQL offers immense power and flexibility, even experienced users can fall into certain pitfalls that lead to incorrect results, inefficient queries, or unnecessary debugging delays. This chapter explores common mistakes in KQL and provides strategies to avoid and resolve them effectively.

Understanding KQL: A Quick Overview

KQL is a read-only request language used to process large datasets. It is often employed in data exploration, monitoring, and troubleshooting scenarios. The key to mastering KQL lies in understanding its syntax, structure, and the best ways to manipulate data. However, even with a clear understanding, common errors can arise when working with complex datasets.

Common Mistakes and How to Avoid Them

1. Misunderstanding Operators

One of the most common pitfalls in KQL is misunderstanding or misusing operators such as `==`, `=~`, and `!`. For instance:

- `==` is used for exact matches.
- `=~` is for case-insensitive comparisons.
- `!` serves as a negation operator.

Example Mistake:

```
| where Name == "value"
```

This query fails for case-insensitive matches, leading to missed results.

Solution:

To match without case sensitivity, use:

```
| where Name =~ "value"
```

2. Overlooking Null or Empty Values

KQL does not implicitly handle null or empty fields in your dataset. Forgetting to account for these can skew results.

Example Mistake:

```
| where ColumnName == "SomeValue"
```

If **ColumnName** contains nulls, this query may miss relevant data.

Solution:

Always include null checks if they are applicable:

```
| where isnotempty(ColumnName) and  
ColumnName == "SomeValue"
```

3. Neglecting Data Types

KQL is type-sensitive, and mismatching data types can lead to errors or unexpected results.

Example Mistake:

Comparing a string field with an integer value:

```
| where NumericField == "123"
```

Solution:

Ensure type compatibility:

```
| where NumericField == 123
```

You can use type conversion functions such as `tostring()` or `toint()` when necessary.

4. Inefficient Use of Filters

Applying filters after heavy computations can lead to excessive resource consumption and slower query performance.

Example Mistake:

```
| extend NewColumn =  
some_function(ColumnName)
```

```
| where NewColumn == "value"
```

Solution:

Always filter data as early as possible:

```
| where ColumnName == "value"
```

```
| extend NewColumn =  
some_function(ColumnName)
```

5. Ignoring Query Limits

KQL has default limits on the number of records and the amount of data a query can return. Failing to understand these limits can lead to incomplete results.

Solution:

Explicitly set limits when you need more data, and optimize queries to retrieve only the required information:

```
| take 10000
```

6. Using Wildcards Incorrectly

Wildcards in KQL are powerful but must be used cautiously. Misplacing * or using it excessively can lead to inefficient or incorrect queries.

Example Mistake:

```
| where ColumnName matches regex ".*value.*"
```

This can be computationally expensive and may not always provide precise results.

Solution:

Use functions like **contains** or **startswith** for better performance:

```
| where ColumnName contains "value"
```

7. Overcomplicating Queries

Adding unnecessary complexity to a query can make it harder to debug and maintain. For instance, chaining too many operations without clarity or reusing similar logic redundantly.

Solution:

Break queries into logical steps and use comments to document their purpose. This approach not only aids debugging but also enhances collaboration.

8. Forgetting to Use Summarization Functions

Failure to utilize summarization functions like **summarize** and **count** can result in manually interpreting raw data, which is error-prone.

Example Mistake:

Manually looking for distinct values instead of summarizing them:

```
| distinct ColumnName
```

Solution:

Use summarization for clarity and efficiency:

```
| summarize CountByValue = count() by  
ColumnName
```

9. Skipping Validation of Results

Even if a query runs successfully, its results may not always be correct. Overlooking validation, especially with complex conditions, can lead to incorrect conclusions.

Solution:

Cross-check intermediate results using smaller datasets and basic queries before scaling them up.

10. Ignoring Case Sensitivity in String Operations

KQL operations on strings are case-sensitive by default, which may lead to unintended mismatches or exclusions.

Example Mistake:

```
| where Name == "value"
```

Solution:

If case insensitivity is required, always opt for `=~`:

```
| where Name =~ "value"
```

Debugging Strategies

1. Start Small and Build Incrementally

Avoid writing long, complex queries in one go. Instead, build queries incrementally, validating results at each step. This reduces the risk of errors and makes debugging easier.

2. Use the Query Execution Plan

Leverage tools like the query execution plan to understand how KQL processes your query. Identifying bottlenecks becomes straightforward with this insight.

3. Log Intermediate Results

Break down your query into smaller parts and analyze intermediate results to ensure that each stage behaves as expected.

4. Leverage Built-in Functions

KQL offers several built-in functions for debugging, such as **print** to test expressions or **range** for generating datasets.

5. Format and Document Queries

Always use proper formatting and include comments in your queries. This helps you and others understand the logic behind the query, simplifying debugging in the long run.

Avoiding common mistakes in KQL is not just about learning the syntax but also about adopting a methodical approach to query-writing and debugging. By understanding the typical pitfalls and employing the strategies outlined in this post, you can significantly enhance your efficiency and accuracy when working with KQL. Remember, the key to mastering KQL lies in continuous practice and careful evaluation of your results.

Chapter 51

Integrating KQL with Power Automate & PowerShell: Automating Workflows with Query-Based Decision Logic

The world of automation is evolving rapidly, enabling individuals and organizations to streamline workflows and boost efficiency. One area gaining traction is the integration of KQL (Kusto Query Language) with automation tools like Power Automate and PowerShell. KQL, renowned for its intuitive querying capabilities in Azure Data Explorer and Log Analytics, provides a powerful way to work with data. By integrating it into Power Automate workflows and PowerShell scripts, users can unlock query-based decision logic for dynamic results and actions.

This chapter explores the integration of KQL with Power Automate and PowerShell, providing working examples to help you bring query-based logic into your automation workflows.

Why Integrate KQL with Power Automate and PowerShell?

Integrating KQL with Power Automate and PowerShell combines the strengths of querying and scripting for advanced workflows. Here are a few benefits:

- **Dynamic Decisions:** Automate actions based on real-time data queries.
- **Streamlined Monitoring:** Use KQL to fetch insights from logs and initiate automated responses through Power Automate.
- **Enhanced Productivity:** Leverage PowerShell scripts to perform tasks based on KQL query outputs.
- **Custom Logic:** Combine KQL's query capabilities with the versatility of automation tools for tailor-made solutions.

Integrating KQL with Power Automate

Power Automate enables users to automate workflows across applications and services. By integrating KQL, you can introduce data-driven logic into your workflows.

Step-by-Step Guide

1. **Set up Azure Log Analytics:** Ensure your data is ingested into Azure Log Analytics or Azure Data Explorer. This will serve as the source for KQL queries.
2. **Create a Power Automate Flow:** Log in to Power Automate and create a new flow. You can use triggers like "When a HTTP request is received" or "Scheduled" to start the flow.
3. **Use the Azure Log Analytics Connector:** Add the "Azure Log Analytics" action to your flow. Configure it with your workspace ID and key.
4. **Enter KQL Query:** Input your KQL query to fetch data based on your requirements. For example:
 5. SecurityEvent
 6. | where TimeGenerated > ago(1h)

```
7. | summarize Count = count() by EventID  
  
| where Count > 100
```

This query checks for Event IDs with more than 100 occurrences in the last hour.

8. **Add Conditional Logic:** Use the query results to define actions. For instance, if the "Count" exceeds a threshold, trigger an email alert or start another workflow.

Example Workflow

Imagine you want to monitor failed login attempts and notify administrators accordingly. Here's how it works:

- **Trigger:** A scheduled flow runs every hour.
- **Action 1:** Use KQL to query failed login attempts:

SecurityEvent

```
| where EventID == 4625
```

```
| summarize Count = count() by Account  
  
| where Count > 5
```

- **Action 2:** Send an email notification if Count exceeds 5.

This workflow ensures real-time monitoring of security events, providing proactive responses.

Integrating KQL with PowerShell

PowerShell is a versatile scripting tool ideal for automating administrative tasks. By incorporating KQL, you can create scripts that dynamically respond to query results.

Setting Up

1. Install Azure Data Explorer

Module: Open PowerShell and install the module using:

```
Install-Module -Name Az.Kusto
```

2. **Authenticate:** Use service principal or user credentials to connect to your Azure Data Explorer cluster.
3. **Run KQL Queries:** Write KQL queries within your script using the *Invoke-KustoQuery* function.

Example Script

Suppose you want to clean up resources based on data insights. Here's a sample PowerShell script with an embedded KQL query:

```
# Connect to Azure Data Explorer

$Cluster = "https://.kusto.windows.net"

$Database = ""

$Query = @'

Resources
```

```
| where Status == "Inactive"

| project ResourceID, Name

'@

# Execute KQL query

$Results = Invoke-KustoQuery -Cluster $Cluster
-Database $Database -Query $Query

# Iterate through results and perform cleanup

foreach ($Resource in $Results) {

    Write-Host "Cleaning up resource:
    $($Resource.Name)"
```

```
Remove-AzResource -ResourceId  
$Resource.ResourceID -Force  
  
}
```

This script queries inactive resources and removes them automatically, ensuring efficient resource management.

Combining Power Automate and PowerShell

To maximize automation, you can combine Power Automate and PowerShell. For instance:

- Power Automate initiates a workflow based on KQL queries.
- The flow calls a PowerShell script to execute complex tasks.

Example: Incident Response

1. Power Automate queries security logs for anomalies using KQL.
2. If anomalies are detected, the flow triggers a PowerShell script to isolate

affected systems and notify stakeholders.

Integrating KQL with Power Automate and PowerShell opens up new possibilities for creating intelligent, data-driven automation workflows. From monitoring logs to cleaning resources and responding to incidents, the synergy between these tools transforms how we handle operational challenges. With the examples provided, you can start building powerful workflows tailored to your unique needs.

Whether you're an IT administrator or a data analyst, the combination of KQL, Power Automate, and PowerShell is a game-changer for automation and decision-making. Start exploring today and elevate your automation strategy to the next level!

Chapter 52

Time-Series Analysis with KQL: Unlocking Trends and Forecasting in Azure

Time-series analysis is a powerful technique for understanding trends, detecting anomalies, and forecasting future values based on historical data. In Azure, Kusto Query Language (KQL) provides robust tools like `make-series` and `series_fit` to perform time-series analysis efficiently. This chapter explores KQL's time-series capabilities, demonstrating how to leverage these functions for trend analysis and forecasting in Azure Data Explorer (ADX) or other KQL-supported services like Azure Monitor.

What is Time-Series Analysis?

Time-series data consists of observations collected at regular time intervals, such as CPU usage, website traffic, or sensor readings. Time-series analysis helps uncover patterns (e.g., trends, seasonality) and predict future values. KQL's time-series functions simplify these tasks by transforming raw data into aggregated series and applying mathematical models for analysis.

Key KQL Time-Series Functions

KQL offers several functions tailored for time-series analysis. The most important ones include:

- **make-series**: Aggregates data into a time-series format, creating a sequence of values over fixed time intervals.
- **series_fit_line**: Fits a linear regression line to a time-series to identify trends.
- **series_fit_2lines**: Detects changes in trends by fitting two linear regression lines.
- **series_periods_detect**: Identifies seasonality or periodic patterns in the data.
- **series_outliers**: Detects anomalies in a time-series.
- **series_decompose**: Separates a time-series into trend, seasonal, and residual components.

In this post, we'll focus on make-series and series_fit functions to perform trend analysis and forecasting.

Setting Up the Environment

To follow along, you'll need access to an Azure Data Explorer cluster or a KQL-compatible service

like Azure Monitor. For this example, we'll use a sample dataset containing server performance metrics (e.g., CPU usage) stored in a table called ServerMetrics. The table has columns for Timestamp, ServerName, and CPUUsage.

Step 1: Creating a Time-Series with make-series

The make-series operator aggregates data into a time-series by grouping values over fixed time intervals. Let's create a time-series of average CPU usage per hour for a specific server.

```
ServerMetrics
```

```
| where ServerName == "Server01"
```

```
| make-series AvgCPU = avg(CPUUsage) default=0  
on Timestamp from ago(7d) to now() step 1h
```

Explanation:

- **make-series:** Creates a time-series.

- **AvgCPU**: The name of the series, calculated as the average of CPUUsage.
- **default=0**: Fills missing values with 0.
- **on Timestamp**: Specifies the time column.
- **from ago(7d) to now()**: Defines the time range (last 7 days).
- **step 1h**: Sets the time interval to 1 hour.

Output:

The query returns a single row with two columns:

- **Timestamp**: An array of timestamps (e.g., [2025-05-26T00:00:00Z, 2025-05-26T01:00:00Z, ...]).
- **AvgCPU**: An array of average CPU usage values (e.g., [45.2, 50.1, ...]).

This time-series is ready for further analysis.

Step 2: Trend Analysis with `series_fit_line`

To identify the overall trend in CPU usage, we can use `series_fit_line`, which fits a linear regression line to the time-series.

```
ServerMetrics
```

```
| where ServerName == "Server01"  
  
| make-series AvgCPU = avg(CPUUsage) default=0  
on Timestamp from ago(7d) to now() step 1h  
  
| extend Trend = series_fit_line(AvgCPU)
```

Explanation:

- **series_fit_line(AvgCPU)**: Fits a linear regression line to the AvgCPU series.
- The output includes the slope, intercept, and fitted values, which describe the trend.

Output:

The Trend column contains a dynamic object with properties like:

- **slope**: Indicates the direction and steepness of the trend (positive = increasing, negative = decreasing).

- intercept: The y-intercept of the regression line.
- line_fit: An array of fitted values for visualization.

To visualize the trend, you can plot the AvgCPU and Trend.line_fit arrays in ADX's built-in charting tools.

Step 3: Detecting Trend Changes with series_fit_2lines

If you suspect a change in the trend (e.g., a sudden increase in CPU usage), use series_fit_2lines to fit two linear regression lines and identify the breakpoint.

ServerMetrics

```
| where ServerName == "Server01"
```

```
| make-series AvgCPU = avg(CPUUsage) default=0  
on Timestamp from ago(7d) to now() step 1h
```

```
| extend TwoTrends = series_fit_2lines(AvgCPU)
```

Explanation:

- **series_fit_2lines(AvgCPU)**: Fits two regression lines and detects the point where the trend changes.

Output:

The TwoTrends column includes:

- **breakpoint**: The index in the series where the trend changes.
- **line1_fit** and **line2_fit**: Arrays of fitted values for the two lines.

This is useful for pinpointing events like system upgrades or failures that alter resource usage patterns.

Step 4: Forecasting with Linear Regression

While KQL doesn't have a dedicated forecasting function, you can extend the linear regression from `series_fit_line` to predict future values.

Here's an example:

```
| where ServerName == "Server01"

| make-series AvgCPU = avg(CPUUsage) default=0
on Timestamp from ago(7d) to now() step 1h

| extend Trend = series_fit_line(AvgCPU)

| project Timestamp, AvgCPU, TrendSlope =
Trend.slope, TrendIntercept = Trend.intercept

| extend Forecast = TrendSlope *
array_length(Timestamp) + TrendIntercept
```

Explanation:

- **Trend.slope and Trend.intercept:**
Extracted from series_fit_line.
- **array_length(Timestamp):** Represents the next time step for forecasting.
- **Forecast:** Calculates the predicted value for the next hour.

This simple approach assumes the trend continues linearly. For more complex forecasting,

consider combining KQL with Azure Machine Learning.

Step 5: Visualizing Results

Azure Data Explorer's visualization pane lets you plot time-series data. To visualize the actual CPU usage and the fitted trend:

```
ServerMetrics
```

```
| where ServerName == "Server01"
```

```
| make-series AvgCPU = avg(CPUUsage) default=0  
on Timestamp from ago(7d) to now() step 1h
```

```
| extend Trend = series_fit_line(AvgCPU)
```

```
| project Timestamp, AvgCPU, TrendLine =  
Trend.line_fit
```

```
| render timechart
```

This query generates a line chart with AvgCPU and TrendLine, making it easy to spot trends.

Practical Applications

KQL's time-series capabilities are invaluable for:

- **Monitoring:** Track resource usage (e.g., CPU, memory) and detect anomalies.
- **Capacity Planning:** Forecast future resource needs based on historical trends.
- **Anomaly Detection:** Identify unusual spikes or drops in metrics.
- **Business Insights:** Analyze time-based KPIs like website traffic or sales.

Tips for Effective Time-Series Analysis

1. **Choose the Right Time Interval:** Use a step size in make-series that matches your analysis needs (e.g., 1m for real-time monitoring, 1d for long-term trends).

2. **Handle Missing Data:** Use the default parameter in make-series to fill gaps appropriately.
3. **Combine with Other Functions:** Use series_outliers or series_periods_detect for deeper insights.
4. **Scale with ADX:** For large datasets, leverage ADX's distributed architecture to process millions of records quickly.

KQL's time-series functions, such as make-series and series_fit, provide a powerful and accessible way to perform trend analysis and forecasting in Azure. By transforming raw data into time-series and applying regression models, you can uncover insights and make data-driven decisions.

Whether you're monitoring infrastructure, analyzing business metrics, or predicting future trends, KQL and Azure Data Explorer are your go-to tools for time-series analysis.

Chapter 53

KQL Mythbusters: Debunking Common Misconceptions

In the world of data querying, Kusto Query Language (KQL) often flies under the radar, especially outside Microsoft Azure ecosystems. KQL is a powerful, read-only query language designed for analyzing large volumes of structured, semi-structured, and unstructured data—think logs, telemetry, metrics, and time-series information. It's used in tools like Azure Data Explorer, Azure Monitor, Microsoft Sentinel, and Microsoft Fabric. Despite its growing popularity, several myths persist that can deter newcomers from exploring its potential. In this post, we'll bust some of the most common misconceptions about KQL, using simple examples to highlight its accessibility and versatility. Whether you're a data analyst, developer, or IT professional, you'll see why KQL deserves a spot in your toolkit.

To demonstrate, we'll use the StormEvents sample dataset, [a table available in Azure Data Explorer demos](#). It contains real-world data on storm events in the US from 2007, with columns like StartTime (datetime), EndTime (datetime), State (string), EventType (string), DamageProperty (integer in dollars), and more. Imagine it as a table with about 59,066 rows of weather-related incidents.

Myth 1: KQL Is Only for Azure Experts

One prevalent myth is that KQL is exclusively for seasoned Azure professionals—those deeply embedded in cloud architecture or DevOps. In reality, KQL is accessible to anyone with basic querying experience, even if you're new to Azure. It's designed to be intuitive and doesn't require extensive knowledge of Azure services to get started. You can run KQL queries in free demo environments like the Azure Data Explorer web UI without any setup.

Debunking with an Example

Let's start with a basic query to count all storm events in the dataset. This is as simple as it gets—no Azure expertise needed.

KQL Query:

```
StormEvents
```

```
| count
```

Explanation: This query references the StormEvents table and uses the count operator to tally the rows. The pipe (|) symbol chains operations, making it readable like a flowchart.

Result:

See? That's the total number of storm events recorded. If you've ever used SQL's SELECT COUNT(*) FROM table, this is strikingly similar but more streamlined. Beginners can experiment

with this in minutes using Microsoft's free KQL playground.

Myth 2: KQL Is Too Complex for Beginners

Another common misunderstanding is that KQL's syntax is overly complicated, reserved for advanced users. On the contrary, KQL is engineered for simplicity and readability. Its pipe-based structure (inspired by Unix pipelines) makes queries flow logically from left to right, reducing verbosity compared to traditional SQL. It's case-sensitive for clarity, and most everyday queries use just a handful of operators like where (filter), project (select columns), and summarize (aggregate).

Debunking with an Example

Here's a beginner-friendly query to preview a small sample of data and filter it.

KQL Query:

```
StormEvents
```

```
| take 5  
  
| project State, EventType, DamageProperty
```

Explanation:

- take 5 grabs 5 arbitrary rows (like SQL's TOP 5 for previewing).
- project selects only the specified columns, keeping the output clean.

Result (Sample Output):

State	EventType	DamageProperty
ATLANTIC SOUTH	Waterspout	0
FLORIDA	Heavy Rain	0
FLORIDA	Tornado	6200000
GEORGIA	Thunderstorm Wind	2000
MISSISSIPPI	Thunderstorm Wind	20000

This query is straightforward—no nested subqueries or complex joins required. For filtering, add a where clause:

Enhanced Query:

```
StormEvents
```

```
| where State == "TEXAS" and EventType ==  
"Flood"
```

```
| take 5
```

```
| project StartTime, State, EventType,  
DamageProperty
```

Result (Sample Output):

StartTime	State	EventType	DamageProperty
2007-01-13T08:45:00Z	TEXAS	Flood	0
2007-01-13T09:30:00Z	TEXAS	Flood	0
2007-03-13T16:00:00Z	TEXAS	Flood	50000
2007-04-24T16:30:00Z	TEXAS	Flood	100000
2007-05-06T00:00:00Z	TEXAS	Flood	250000

Beginners can build on this incrementally, adding operators as needed. KQL's error messages are helpful, and resources like Microsoft's tutorials make learning a breeze.

Myth 3: KQL Is Limited to Specific Applications Like Microsoft Sentinel

A frequent misconception is that KQL is niche, primarily for security tools like Microsoft Sentinel. While it's excellent for log analysis in Sentinel, KQL's versatility extends far beyond. It's used across Azure services for everything from monitoring application performance in Azure Monitor to exploring big data in Azure Data Explorer. It handles time-series analysis, geospatial queries, and even machine learning integrations seamlessly.

Debunking with an Example

To show versatility, let's aggregate data to find unique event types and sort by property damage—useful for analytics in any domain.

KQL Query:

```
StormEvents
```

```
| distinct EventType
```

Explanation: The distinct operator lists unique values in the EventType column, revealing the variety of data KQL can handle (e.g., from logs to metrics).

Result (Partial Output, 46 Unique Types):

EventType
Thunderstorm Wind
Hail
Flash Flood
Drought
Winter Weather
...

Now, for aggregation and sorting:

KQL Query:

```
StormEvents
```

```
| where State == "TEXAS"
```

```
| summarize TotalDamage = sum(DamageProperty)  
by EventType  
  
| sort by TotalDamage desc
```

Explanation:

- summarize groups by EventType and calculates the sum of DamageProperty.
- sort by orders results descending.

Result (Sample Output):

EventType	TotalDamage
Flood	15000000
Tornado	10000000
Thunderstorm Wind	5000000
Hail	2000000
...	

This could apply to business intelligence (e.g., sales data) or IoT metrics, not just security logs. KQL's operators like join for combining datasets or make-series for time-series further prove its broad applicability.

Myth 4: KQL Is Inferior to SQL and Not Worth Learning

Some believe KQL pales in comparison to SQL, viewing it as a "lite" version without the power for serious work. Actually, KQL builds on SQL's strengths while addressing pain points—it's less verbose, optimized for big data, and excels in areas like text search and anomaly detection where SQL might struggle. It's not a replacement but a complement, especially for interactive, exploratory querying.

Debunking with an Example

Compare a simple SQL query to its KQL equivalent for filtering and joining. Suppose we join StormEvents with a hypothetical StatesInfo table (with columns State and Population).

SQL Equivalent (Hypothetical):

```
SELECT se.State, se.EventType, si.Population  
FROM StormEvents se
```

```
INNER JOIN StatesInfo si ON se.State = si.State  
  
WHERE se.DamageProperty > 100000  
  
ORDER BY se.State;
```

KQL Query:

```
StormEvents  
  
| where DamageProperty > 100000  
  
| join kind=inner (  
  
    StatesInfo  
  
) on State  
  
| project State, EventType, Population
```

```
| sort by State
```

Explanation: KQL's pipe flow makes it concise and easy to debug. It supports all major join types and handles large datasets efficiently.

KQL shines in modern scenarios like parsing JSON logs or detecting patterns in time-series data, often with fewer lines of code than SQL.

Time to Bust Your Own Myths

KQL isn't an elite tool for Azure gurus—it's a user-friendly language that empowers beginners and experts alike to unlock insights from diverse data sources. By debunking these myths with real examples from the StormEvents dataset, we've seen how simple queries can filter, aggregate, and analyze data effortlessly. If you're curious, head to the Azure Data Explorer demo cluster or Microsoft Learn for hands-on practice. Start small, experiment, and you'll quickly appreciate KQL's power. What's stopping you from querying your way to better data decisions?

Chapter 54

Exploring Kusto.Cli: A Command-Line Utility for Kusto Query Language

Kusto.Cli is a powerful command-line utility designed to interact with Kusto clusters, enabling users to send queries and control commands efficiently. This tool is particularly useful for automating tasks that typically require writing code, such as C# programs or PowerShell scripts. In this chapter, we'll explore the various modes, features, and commands of Kusto.Cli, providing a comprehensive guide to harnessing its capabilities.

Modes of Operation

Kusto.Cli operates in three primary modes:

1. **REPL Mode:** This mode allows users to enter queries and commands interactively. The tool displays the results and awaits the next input,

making it ideal for exploratory data analysis.

2. **Execute Mode:** Users can specify one or more queries and commands as command-line arguments. These are executed sequentially, and the results are output to the console. Optionally, the tool can switch to REPL mode after executing all commands.
3. **Script Mode:** Similar to execute mode, but queries and commands are specified in a script file. This mode is perfect for running predefined sets of commands.

Getting Started

To use Kusto.Cli, download the Microsoft.Azure.Kusto.Tools NuGet package and extract the tools folder to your target directory. No additional installation is required. The basic command to run Kusto.Cli is:

```
Kusto.Cli.exe "<ConnectionString>"
```

Command-Line Arguments

Kusto.Cli supports various command-line arguments to customize its behavior:

- **ConnectionString**: Specifies the Kusto connection string.
- **-execute**: Runs the specified query or command.
- **-script**: Executes commands from a script file.
- **-keepRunning**: Enables or disables REPL mode after executing commands.
- **-echo**: Enables or disables echo mode, repeating queries or commands in the output.
- **-transcript**: Writes program output to a specified file.
- **-logToConsole**: Enables or disables console output.
- **-lineMode**: Determines how newlines are treated in input queries or commands.

Directives

Kusto.Cli includes several directives for controlling its behavior:

- **#help**: Displays a short help message.
- **#quit**: Exits the tool.
- **#connect**: Connects to a different Kusto service.

- **#dbcontext**: Changes the context database.
- **#save**: Saves query results to a CSV file.
- **#script**: Executes a script file.

Example Usage

Here's an example of using Kusto.Cli to export query results to a CSV file:

```
Kusto.Cli.exe "<ConnectionString>" -  
execute:"#save c:\temp\results.csv" -  
execute:"StormEvents | take 10"
```

Kusto.Cli is a versatile tool that simplifies interaction with Kusto clusters through its command-line interface. Whether you're performing exploratory data analysis, automating tasks, or running predefined scripts, Kusto.Cli offers the flexibility and power needed to streamline your workflow. By understanding its modes, command-line arguments, and directives, you can fully leverage Kusto.Cli to enhance your data querying capabilities.

Chapter 55

Mastering Cross-Cluster and Federated Querying in Kusto Query Language (KQL)

In the world of big data analytics, Azure Data Explorer (ADX) stands out for its speed and scalability, powered by the Kusto Query Language (KQL). While KQL is intuitive for querying within a single database or cluster, things get more complex when you need to pull data from multiple clusters or even hybrid environments mixing cloud and on-premises sources. Cross-cluster and federated querying enable seamless data aggregation across boundaries, but they come with challenges like authentication hurdles, performance overhead, and setup intricacies. These topics are often glossed over in basic tutorials, reserved for enterprise-scale setups where data silos are the norm. In this post, we'll dive into the mechanics of querying across Azure Data Explorer clusters using unions and external references, then explore optimizing federated joins for hybrid scenarios—think

combining on-premises logs with cloud-based telemetry.

Whether you're a data engineer dealing with distributed systems or an analyst hunting for insights across environments, understanding these advanced KQL features can unlock powerful analytics. We'll cover syntax, best practices, and pitfalls, drawing from real-world implications.

Cross-Cluster Querying: Unions and External References

At its core, cross-cluster querying lets you reference data outside your current “database in context”—the default scope where your query runs and permissions are checked. This is crucial for scenarios where data is sharded across multiple ADX clusters, perhaps for geographic distribution or workload isolation.

Basic Syntax for Referencing Remote Data

To query a table in another database or cluster, use the `cluster()` and `database()` functions to qualify the entity name. For example:

- Within the same cluster but different database:
database("OtherDatabase").MyTable
- In a remote cluster:
cluster("https://remotecluster.kusto.windows.net").database("RemoteDb").MyTable

This qualified naming extends to functions, views, and other entities. Authentication is key here: you need at least viewer permissions on the default database *and* all referenced databases or clusters. Without proper access, queries will fail with permission errors, making role-based access control (RBAC) setup a prerequisite in enterprise environments.

Leveraging Unions for Multi-Cluster Aggregation

The union operator is your go-to for combining results from multiple tables across clusters. It merges rows from input tables into a single result set, handling schema differences by aligning columns where possible.

A simple example:

```
union
```

```
MyLocalTable,  
  
database("OtherDb").RemoteTable,  
  
cluster("https://anothercluster.kusto.windows.net").database("DbThere").AnotherTable  
  
| where Timestamp > ago(1d)  
  
| summarize Count = count() by Category
```

For broader scopes, use wildcards (but not on cluster names):

```
union withsource=SourceTable *  
  
| union database("OtherDb*").*Table
```

```
| union  
cluster("https://remote.kusto.windows.net").dat  
abase("*").*
```

This is powerful for aggregating logs from regional clusters, but watch for performance: unions can process massive datasets, and default row limits (e.g., 500,000 records) apply unless you set the notruncation option. Schema caching helps efficiency but can cause issues if remote schemas change—clear the cache with .clear cache remote-schema if needed.

External references shine when incorporating non-ADX data. The externaldata operator lets you query CSV, JSON, or other files from storage like Azure Blob or ADLS directly in KQL:

```
externaldata (Timestamp:datetime,  
Message:string)
```

```
[@"https://mystorage.blob.core.windows.net/logs  
/logfile.csv"]
```

```
with (format="csv", ignoreFirstRecord=true)
```

```
| union MyCloudTable  
  
| where Message contains "error"
```

This bridges cloud data with external sources, but for true on-premises integration, you'll often need to set up connectors or ingest data first—more on that in the hybrid section.

Cross-Cluster Joins: Merging Data Across Boundaries

Joins take cross-cluster querying further by correlating datasets, but they introduce federation complexities. A cross-cluster join merges tables from different clusters, with execution potentially “remoted” to optimize data movement.

Syntax and Strategies

The basic join syntax mirrors standard KQL, but with qualified references and an optional `hint.remote` for strategy:

```
LocalTable
```

```
| join hint.remote=right (
```



```
cluster("https://remotecluster.kusto.windows.net")
      .database("RemoteDb")
      .RemoteTable
```



```
) on CommonKey
```

Strategies include:

- auto (default): ADX decides based on data location—often the right cluster if the left is remote.
- left, right, or local: Explicitly choose where to run the join.

For instance, if the remote table is much larger, use `hint.remote=right` to avoid shipping gigabytes across networks.

Authentication follows the same rules: ensure viewer access across all involved clusters.

Performance hits come from data transfer and serialization—joins can be CPU-intensive, so profile with the `set query_trace=true` option to spot bottlenecks.

Optimizing Federated Joins in Hybrid Environments

Hybrid setups—e.g., on-premises logs (like from local servers or legacy systems) joined with cloud data in ADX—are where federated querying gets tricky. “Federated” here means querying without full data movement, often via external tables or cross-service links.

Setting Up for Hybrid Queries

For on-premises data, direct federation isn't always native, but you can:

1. **Ingest on-prem logs into ADX** via tools like Azure Data Factory or Event Hubs for streaming, then query as standard tables.
2. **Use external tables** to reference on-prem SQL databases or files if they're accessible via public endpoints. Create an external table in ADX:

```
.create external table OnPremLogs (Id: long,  
LogMessage: string)
```

```
kind=sql
```

```
(h@"Server=tcp:myserver.database.windows.net,14  
33;Database=MyDb;Authentication=Active  
Directory Integrated")
```

```
with (table="OnPremTable")
```

3. Then join: CloudTable | join kind=inner OnPremLogs on Id
4. **Cross-service querying** with Azure Monitor/Log Analytics: If on-prem logs are routed to Log Analytics (via agents), query them from ADX:

```
workspace ("MyLogAnalyticsWorkspace").Perf
```

```
| join kind=inner MyAdxTable on Computer
```

5. This federates cloud ADX data with on-prem telemetry collected in Log Analytics.

Performance Optimization Tips

- **Filter early:** Apply where clauses right after table references to reduce data scanned—e.g., filter timestamps first, then strings.
- **Choose the right side:** For joins, put the smaller dataset on the left; for cross-cluster, execute on the cluster with the bulk of data using `hint.remote`.
- **Use alternatives:** Swap join for in or lookup when possible—lookup is faster for small right-side datasets.
- **Materialize for reuse:** If subqueries are repeated, use `materialize()` in let statements to cache intermediates.
- **Shuffle for high cardinality:** Add `hint.shufflekey=<key>` to joins or summarizes with many unique keys to distribute processing.
- In hybrid scenarios, minimize network hops—ingest on-prem data to cloud if queries are frequent, as federation adds latency.

These tweaks can slash query times from minutes to seconds, especially in distributed setups where data volumes explode.

Why Master This Now?

Cross-cluster and federated querying in KQL isn't just a nice-to-have—it's essential for modern, distributed analytics in enterprises. While authentication demands careful RBAC planning and performance requires thoughtful optimization, the payoff is unified insights without massive data migrations. Start small: test unions in your dev clusters, then scale to hybrids with external tables or Log Analytics integration.

If you're in an enterprise setup, dive into the docs for your specific auth model (e.g., Entra ID). Experiment, monitor query metrics, and remember: the key to efficiency is reducing data movement.

Chapter 56

Kusto Query Language (KQL) Plugin Integrations for Custom Computations

Kusto Query Language (KQL) is a powerful tool for querying and analyzing large datasets in Azure Data Explorer. While KQL excels at filtering, aggregating, and joining data, its plugin integrations, such as `python()` and `R()`, unlock advanced capabilities by embedding external programming languages for custom computations. These plugins enable sophisticated tasks like machine learning scoring and statistical analysis, which are not commonly covered in standard KQL tutorials. This chapter explores how to leverage the `python()` and `R()` plugins for custom computations, focusing on practical applications like anomaly detection and regression analysis.

The Power of KQL Plugins

KQL plugins allow users to extend KQL's functionality by embedding code in languages

like Python and R directly within queries. This is particularly useful for scenarios requiring advanced computations that go beyond KQL's built-in operators. The `python()` and `R()` plugins enable seamless integration of these languages, allowing users to process query results with custom logic while maintaining the efficiency of KQL's data processing engine.

Key Benefits of Plugins

- **Flexibility:** Run Python or R code inline with KQL queries.
- **Advanced Analytics:** Perform machine learning, statistical modeling, or custom computations on query results.
- **Data Serialization:** Handle data exchange between KQL and external languages efficiently using formats like JSON or tabular data.
- **Scalability:** Leverage Azure Data Explorer's distributed computing for large-scale data processing.

Using the `python()` Plugin for Machine Learning Scoring

The `python()` plugin allows users to embed Python code within KQL queries, making it ideal for tasks

like machine learning scoring. For example, you can apply anomaly detection models to query results to identify unusual patterns in data, such as system logs or IoT telemetry.

Example: Anomaly Detection with `python()`

Suppose you have a dataset of system performance metrics (e.g., CPU usage) and want to detect anomalies using a Python-based isolation forest model. The `python()` plugin can process the query results and return scores indicating potential anomalies.

Sample KQL Query with `python()`

```
let data = MyTable  
  
| where Timestamp > ago(1h)  
  
| project CpuUsage, Timestamp;  
  
data
```

```
| evaluate python(  
  
    typeof(*, AnomalyScore:double),  
  
    ...  
  
from sklearn.ensemble import IsolationForest  
  
import pandas as pd  
  
def evaluate_anomaly(df):  
  
    model = IsolationForest(contamination=0.1)  
  
    scores = model.fit_predict(df[['CpuUsage']])  
  
    df['AnomalyScore'] = scores
```

```
    return df

    result = evaluate_anomaly(df)

    ```

)

| where AnomalyScore == -1
```

## Explanation

- 1. Data Preparation:** The query filters `MyTable` for the last hour of data and projects the `CpuUsage` and `Timestamp` columns.
- 2. Python Plugin:** The `evaluate python()` operator invokes the Python runtime. The `typeof(*, AnomalyScore:double)` specifies that the output includes all input columns (\*) plus a new column `AnomalyScore` of type `double`.

3. **Python Code:** Inside the plugin, a Python script uses `scikit-learn`'s `IsolationForest` to compute anomaly scores. The input data is automatically passed as a pandas DataFrame (`df`), and the script returns the DataFrame with an additional `AnomalyScore` column.
4. **Filtering Anomalies:** The final `where` clause filters for rows where `AnomalyScore == -1`, indicating anomalies.

## Use Case

This approach is valuable for real-time monitoring in scenarios like DevOps, where detecting unusual spikes in CPU or memory usage can trigger alerts. The `python()` plugin enables seamless integration of machine learning models without leaving the KQL environment.

## Notes

- **Dependencies:** Ensure required Python libraries (e.g., `scikit-learn`) are available in the Azure Data Explorer Python sandbox.
- **Performance:** The `python()` plugin runs in a sandboxed environment, so large datasets may require optimization to avoid memory or timeout issues.

- **Serialization:** Input data is serialized as a pandas DataFrame, and the output must match the specified schema.

## Using the `R()` Plugin for Statistical Analysis

The `R()` plugin enables embedding R code within KQL queries, making it ideal for statistical analysis tasks like regression modeling. R is particularly suited for scenarios requiring advanced statistical techniques, such as linear regression or time-series analysis.

### Example: Linear Regression with `R()`

Suppose you have a dataset of sales data and want to perform a linear regression to predict sales based on advertising spend. The `R()` plugin can process the query results and return regression coefficients or predictions.

### Sample KQL Query with `R()`

```
let data = SalesTable

| where Date > ago(30d)
```

```
| project AdvertisingSpend, Sales;

data

| evaluate R(

 typeof(*, PredictedSales:double,
Coefficient:double),

    ~~~  
  
df <- data.frame(df)  
  
model <- lm(Sales ~ AdvertisingSpend, data=df)  
  
df$PredictedSales <- predict(model, df)  
  
df$Coefficient <- coef(model) ["AdvertisingSpend"]
```

```
df  
  
    ...  
  
)  
  
| project Date, AdvertisingSpend, Sales, PredictedSales,  
Coefficient
```

## Explanation

- 1. Data Preparation:** The query filters `SalesTable` for the last 30 days and projects the `AdvertisingSpend` and `Sales` columns.
- 2. R Plugin:** The `evaluate R()` operator invokes the R runtime. The `typeof(*,` `PredictedSales:double,` `Coefficient:double)` specifies the output schema, including all input columns plus `PredictedSales` and `Coefficient`.
- 3. R Code:** The R script converts the input data to a data frame, fits a linear regression model using `lm()`, and computes predicted sales and the

regression coefficient for `AdvertisingSpend`.

The output is returned as a data frame.

4. **Output:** The final `project` clause selects relevant columns for further analysis or visualization.

## Use Case

This approach is useful for business analytics, such as understanding the relationship between marketing spend and sales outcomes.

The `R()` plugin allows analysts to leverage R's statistical capabilities directly within KQL.

## Notes

- **Dependencies:** Ensure R libraries are available in the Azure Data Explorer R sandbox.
- **Data Serialization:** Input data is serialized as a data frame, and the output must conform to the specified schema.
- **Performance:** Similar to the `python()` plugin, large datasets may require optimization to manage memory and computation time.

## Handling Data Serialization

Both the `python()` and `R()` plugins handle data serialization automatically, converting KQL query results into formats compatible with Python (`pandas DataFrame`) or R (`data frame`). However, users must ensure the output schema matches the `typeof` specification to avoid errors. Key considerations include:

- **Input Format:** KQL query results are passed as tabular data, which is converted to a `DataFrame`/`data frame`.
- **Output Format:** The plugin's output must be a `DataFrame`/`data frame` with columns matching the `typeof` declaration.
- **Data Types:** Ensure compatibility between KQL data types (e.g., `double`, `string`) and Python/R data types to prevent serialization issues.
- **Large Datasets:** For large datasets, consider sampling or aggregating data in KQL before passing it to the plugin to reduce memory usage.

## Best Practices for KQL Plugin Integrations

1. **Optimize Data Size:** Use KQL's filtering, aggregation, and sampling operators

- (e.g., `take`, `summarize`) to reduce the dataset size before passing it to the plugin.
2. **Validate Output Schema:** Always define the output schema using `typeof` to ensure compatibility with downstream KQL operations.
  3. **Test Incrementally:** Start with small datasets to debug Python or R code before scaling to larger datasets.
  4. **Leverage Sandbox Libraries:** Use pre-installed libraries in the Azure Data Explorer sandbox to avoid dependency issues.
  5. **Monitor Performance:** Be mindful of timeouts and memory limits in the sandboxed environment, especially for complex computations.

The `python()` and `R()` plugins in KQL open up a world of possibilities for advanced analytics, from machine learning scoring to statistical modeling. By embedding Python or R code within KQL queries, users can perform custom computations like anomaly detection and regression analysis without leaving the Azure Data Explorer environment. These plugins combine the scalability of KQL with the flexibility of Python and R, making them powerful tools for data

scientists and analysts. Whether you're detecting anomalies in system metrics or modeling business trends, KQL plugin integrations provide a seamless way to extend your analytics capabilities.

For more details, check the [Azure Data Explorer documentation](#) or experiment with these plugins in your own KQL queries!

# Chapter 57

---

## Kusto Query Language (KQL) Materialized Views for Performance Optimization

Kusto Query Language (KQL) is a powerful tool for querying and analyzing large datasets in Azure Data Explorer (ADX). One of its standout features for performance optimization is **materialized views**, which allow pre-computed aggregations to significantly reduce query execution times, especially for high-volume logs and streaming data scenarios. However, designing and leveraging materialized views effectively—particularly with real-time update policies and optimized query patterns—remains a complex and under-discussed topic. In this chapter, we'll explore how to create and query materialized views with aggregation policies to optimize performance for large datasets.

### What Are Materialized Views in KQL?

Materialized views in KQL are pre-computed, persisted query results that store aggregated or

transformed data from a source table. Unlike regular views, which are virtual and computed on-the-fly, materialized views physically store the results, making them ideal for scenarios where you need to repeatedly query aggregated data over large datasets. They are particularly useful for:

- Reducing scan times by avoiding repetitive calculations on raw data.
- Enabling real-time analytics on streaming data with minimal latency.
- Simplifying complex queries by pre-aggregating data.

Materialized views are updated automatically based on an **update policy**, which defines how and when the view is refreshed with new data from the source table.

## Designing Update Policies for Real-Time Materialized Views

In streaming data scenarios, where data arrives continuously (e.g., IoT telemetry, application logs, or event streams), materialized views must balance performance with data freshness.

The **update policy** determines how the view

processes incoming data, and designing it correctly is critical for real-time analytics.

## Key Considerations for Update Policies

### 1. Source Table and Ingestion:

- Materialized views are tied to a single source table. Ensure the source table is optimized for ingestion (e.g., using partitioning or sharding for high-volume data).
- For streaming data, use ingestion methods like Event Hubs or Kafka to ensure low-latency data arrival.

### 2. Update Frequency:

- The update policy defines a **time window** for processing new data (e.g., every 5 minutes). Smaller windows improve data freshness but increase computational overhead.
- Example: For a materialized view summarizing IoT sensor data, you might set a 1-minute window to ensure near-real-time insights.

### 3. Aggregation Scope:

- Specify which columns to aggregate (e.g., `sum`, `count`, `avg`) and how to group them (e.g., `by device ID` or `time bin`).
- Use `materialize()` to define the aggregation logic, ensuring it aligns with your query patterns.

### 4. Backfill Strategy:

- For historical data, configure the update policy to backfill the materialized view with existing data. This is crucial when initializing a view or recovering from failures.
- Example: Set `BackfillFrom` to a specific timestamp to process historical data up to the current time.

## Example: Creating a Materialized View with an Update Policy

Suppose you have a source table `SensorData` with columns `Timestamp`, `DeviceId`, `Temperature`, and `Humidity`. You want to create a materialized view that computes the average temperature per device every 5 minutes for real-time monitoring.

```
.create materialized-view SensorAvgTemps on  
table SensorData
```

```
{
```

```
SensorData
```

```
| summarize AvgTemp = avg(Temperature) by  
DeviceId, bin(Timestamp, 5m)
```

```
}
```

```
with (
```

```
updateInterval = 5m,
```

```
backfillFrom = datetime(2025-09-01),
```

```
updateMode = 'Incremental'
```

)

- `updateInterval`: Refreshes the view every 5 minutes.
- `backfillFrom`: Processes historical data starting from September 1, 2025.
- `updateMode`: `Incremental` ensures only new data is processed, reducing overhead.

This materialized view pre-computes the average temperature, allowing queries to run against a smaller, aggregated dataset instead of scanning the entire `SensorData` table.

## Best Practices for Update Policies

- **Minimize Update Frequency:** Balance freshness with performance. For example, a 1-minute interval may be too aggressive for low-priority analytics.
- **Monitor Resource Usage:** Frequent updates on large datasets can strain cluster resources. Use `.show materialized-view` to monitor performance and adjust as needed.
- **Handle Data Gaps:** Use `lookback` in the update policy to reprocess data if ingestion delays occur (e.g., `lookback = 1h` to account for late-arriving data).

# Query Patterns to Leverage Materialized Views

Materialized views shine when you optimize query patterns to take advantage of their pre-aggregated data. Below are strategies to reduce scan times in high-volume log scenarios, such as analyzing application logs or telemetry data.

## 1. Query the Materialized View Directly

Instead of querying the raw source table, target the materialized view to avoid scanning large datasets. For example, to get the average temperature per device from the `SensorAvgTemps` view:

```
SensorAvgTemps
```

```
| where Timestamp > ago(1h)
```

```
| summarize MaxAvgTemp = max(AvgTemp) by DeviceId
```

This query scans the pre-aggregated view, which is significantly smaller than the `raw SensorData` table, reducing execution time.

## 2. Use Time Filters Efficiently

Materialized views are optimized for time-based queries. Always include a time filter (e.g., `where Timestamp > ago(1d)`) to limit the data scanned, as materialized views are often partitioned by time.

```
SensorAvgTemps
```

```
| where Timestamp between (datetime(2025-09-24 00:00:00) .. datetime(2025-09-24 23:59:59))
```

```
| summarize AvgTemp = avg(AvgTemp) by DeviceId
```

## 3. Combine with Joins for Complex Analysis

If you need to combine aggregated data with other tables, use the materialized view in joins to minimize the data processed. For example, joining `SensorAvgTemps` with a `DeviceMetadata` table:

```
SensorAvgTemps
```

```
| join kind=inner DeviceMetadata on DeviceId  
| where AvgTemp > 30  
| project DeviceId, DeviceName, AvgTemp
```

This query leverages the pre-aggregated view to reduce the join's computational cost.

## 4. Avoid Over-Aggregation

While materialized views pre-aggregate data, avoid redundant aggregations in your queries. For instance, if the view already computes `avg(Temperature)`, don't recompute it unless necessary.

## 5. Monitor Query Performance

Use `.show queries` to analyze query execution times and verify that the materialized view is reducing scan times. If performance is suboptimal, check the view's aggregation logic or update policy.

# Challenges and Limitations

While materialized views are powerful, they come with challenges:

- **Storage Overhead:** Materialized views store pre-computed data, increasing storage costs. Ensure the view's scope is focused to avoid unnecessary data retention.
- **Update Latency:** Even with real-time policies, there may be a slight delay (seconds to minutes) between data ingestion and view updates.
- **Complexity in Maintenance:** Changing the aggregation logic or update policy requires recreating the view, which can be disruptive for large datasets.

Materialized views in KQL are a game-changer for optimizing performance in large-scale, high-volume data scenarios. By carefully designing update policies for real-time streaming data and leveraging query patterns that target pre-aggregated views, you can significantly reduce scan times and improve query performance. Start by identifying repetitive query patterns in your workload, create targeted materialized views, and

fine-tune their update policies to balance freshness and resource usage. With these strategies, you'll unlock the full potential of KQL for real-time analytics on massive datasets.

For more details on KQL materialized views, check the [Azure Data Explorer documentation](#).

# Chapter 58

---

## Advanced Summarization in Kusto Query Language (KQL) for Security Analytics in Microsoft Sentinel

Kusto Query Language (KQL) is a powerful tool for analyzing large datasets in security analytics, particularly within Microsoft Sentinel. Advanced summarization techniques in KQL, such as using `make_set` for aggregating unique values and conditional extensions for IP-based summaries, enable security analysts to derive actionable insights efficiently. This chapter explores these techniques, focusing on avoiding query loops and leveraging ASIM (Advanced Security Information Model) schemas for streamlined analysis.

## Understanding ASIM Schemas in Microsoft Sentinel

ASIM schemas provide a standardized framework for normalizing security data in Microsoft Sentinel, ensuring consistency across diverse data sources. For example, ASIM schemas

like `ASimNetworkSession` normalize fields such as source/destination IPs, ports, and event types, making it easier to query heterogeneous data without complex joins or loops. Using ASIM schemas simplifies summarization by providing a unified structure, reducing the need for repetitive preprocessing.

## Using `make_set` for Aggregating Unique Values

The `make_set` operator is ideal for aggregating unique values into an array, which is particularly useful for summarizing attributes like user agents or regions in security logs. By setting sample limits, analysts can control the output size to focus on the most relevant data.

### Example: Aggregating Unique User Agents

Suppose you want to summarize unique user agents per source IP in network session logs. The following KQL query uses `make_set` with a limit to avoid overwhelming results:

```
ASimNetworkSession
```

```
| where EventType == "Traffic"  
  
| summarize UserAgents = make_set(UserAgent,  
10) by SrcIpAddr  
  
| where array_length(UserAgents) > 0
```

In this query:

- `make_set(UserAgent, 10)` aggregates up to 10 unique user agents per source IP.
- The `where` clause filters out empty arrays to keep results clean.
- Using `ASimNetworkSession` ensures normalized fields, avoiding manual mapping of user agent fields across data sources.

This approach is efficient, avoids loops, and provides a concise summary of user agent diversity, which can help identify anomalous patterns (e.g., a single IP using multiple user agents).

## Conditional Extensions for IP-Based Summaries

IP-based summarization often involves metrics like byte counts, unique ports, and directionality (inbound/outbound). Conditional extensions in KQL, such as `extend` with `case` or `iif`, allow dynamic calculations based on specific conditions without iterative loops.

## **Example: Summarizing IP Traffic with Byte Counts and Directionality**

Consider a scenario where you need to summarize bytes transferred, unique ports, and traffic direction for each source IP. The following query demonstrates this:

```
ASimNetworkSession
```

```
| where EventType == "Traffic"
```

```
| extend Direction = case(DstIpAddr ==  
"0.0.0.0", "Outbound", "Inbound")
```

```
| summarize
```

```
TotalBytes = sum(BytesTransferred),  
  
UniquePorts = dcount(DstPort),  
  
Directions = make_set(Direction)  
  
by SrcIpAddr  
  
| where TotalBytes > 0
```

Here's a breakdown:

- `extend Direction = case(...)` **dynamically assigns “Inbound” or “Outbound” based on the destination IP (e.g., “0.0.0.0” indicates outbound traffic in some schemas).**
- `summarize aggregates:`
  - `TotalBytes with sum(BytesTransferred)` **for total data volume.**
  - `UniquePorts with dcount(DstPort)` **for distinct destination ports.**

- Directions with `make_set(Direction)` to capture unique traffic directions.
- Filtering with `where TotalBytes > 0` ensures meaningful results.

This query avoids loops by leveraging KQL's columnar processing and ASIM's normalized fields, making it efficient for large datasets.

## Best Practices for Advanced Summarization

1. **Leverage ASIM Schemas:** Always use ASIM schemas to normalize data upfront, reducing the need for custom joins or field mappings.
2. **Limit Aggregation Scope:** Use parameters like `make_set(..., 10)` to cap the number of unique values and prevent performance degradation.
3. **Avoid Query Loops:** Instead of iterative queries, use `summarize` and `extend` to perform calculations in a single pass.
4. **Filter Early:** Apply `where` clauses before summarization to reduce the dataset size and improve query performance.
5. **Validate Output:** Ensure summarized results are actionable by filtering out

empty or irrelevant aggregations  
(e.g., `array_length(UserAgents) > 0`).

Advanced summarization in KQL, tailored for Microsoft Sentinel, empowers security analysts to extract meaningful insights from complex datasets. By using `make_set` for unique value aggregation and conditional extensions for IP-based metrics, analysts can efficiently summarize data while avoiding query loops. Leveraging ASIM schemas further streamlines the process, ensuring consistency and scalability. These techniques enable faster detection of anomalies, such as unusual user agents or high-volume IP traffic, enhancing security analytics workflows.

# Chapter 59

---

## Using KQL for Time-Delta and Pattern Matching in Identity Attack Detection

The Kusto Query Language (KQL) is a powerful tool for analyzing large datasets, particularly in security contexts like detecting identity attacks such as password sprays or brute-force attempts. By leveraging KQL's time-delta calculations and pattern matching capabilities, analysts can identify suspicious login patterns with temporal logic. This chapter explores advanced techniques using `datetime_diff`, `next`, and statistical aggregations, with results stored in JSON arrays for flexible downstream processing.

## Understanding Identity Attacks

Password spray and brute-force attacks involve repeated login attempts to compromise accounts. Password sprays use a single password across multiple accounts, while brute-force attacks target a single account with many passwords. Both exhibit temporal patterns, such as rapid or

periodic login attempts, which KQL can detect by analyzing time differences and event sequences.

## Time-Delta Analysis with `datetime_diff` and `next`

KQL's `datetime_diff` function calculates the time difference between two timestamps, enabling analysis of inter-event gaps. Combined with the `next` operator, it helps track the time between consecutive login attempts, which is critical for identifying attack patterns.

### Example: Calculating Time Gaps

Suppose you have a log table `SignInLogs` with columns `Timestamp`, `UserPrincipalName`, and `ResultType` (indicating success or failure). To detect rapid login attempts, calculate the time difference between consecutive failed logins for each user.

```
SignInLogs
```

```
| where ResultType == "Failed"
```

```
| sort by UserPrincipalName, Timestamp asc  
  
| extend TimeDelta = datetime_diff('second',  
Timestamp, next(Timestamp, UserPrincipalName))  
  
| where TimeDelta < 60  
  
| summarize AttemptCount = count(),  
MinTimeDelta = min(TimeDelta), MaxTimeDelta =  
max(TimeDelta) by UserPrincipalName
```

## **Explanation:**

- `sort by UserPrincipalName, Timestamp asc`: Orders events by user and time to ensure `next` retrieves the subsequent event for the same user.
- `datetime_diff('second', Timestamp, next(Timestamp, UserPrincipalName))`: Computes the time difference (in seconds) between the current and next login attempt for the same user.
- `where TimeDelta < 60`: Filters for gaps less than 60 seconds, indicating rapid attempts.

- `summarize`: Aggregates the count of attempts and the min/max time deltas per user to identify suspicious patterns.

This query flags users with frequent login attempts, a hallmark of brute-force or password spray attacks.

## Pattern Matching for Attack Detection

KQL's pattern matching capabilities, such as `series_stats` or custom aggregations, can further refine detection by identifying consistent or anomalous time-delta patterns.

### Example: Detecting Periodic Patterns

Password sprays often involve periodic attempts to avoid detection. To identify users with consistent inter-event gaps:

```
SignInLogs
```

```
| where ResultType == "Failed"
```

```
| sort by UserPrincipalName, Timestamp asc

| extend TimeDelta = datetime_diff('second',
Timestamp, next(Timestamp, UserPrincipalName))

| where TimeDelta != null

| summarize TimeDeltas = make_list(TimeDelta)
by UserPrincipalName

| extend Stats = series_stats(TimeDeltas)

| project UserPrincipalName, TimeDeltas,
AvgTimeDelta = Stats.avg, StdDevTimeDelta =
Stats.stdev

| where StdDevTimeDelta < 10 and AvgTimeDelta <
300
```

## Explanation:

- `make_list(TimeDelta)`: Creates an array of time deltas for each user.
- `series_stats(Timedeltas)`: Computes statistical metrics like average and standard deviation of time deltas.
- `where StdDevTimeDelta < 10 and AvgTimeDelta < 300`: Flags users with consistent (low standard deviation) and rapid (low average) time deltas, suggesting automated attack patterns.

## Storing Results in JSON Arrays

Storing results in JSON arrays enables flexible downstream processing, such as integration with alerting systems or machine learning pipelines. KQL's `make_list` and `tojson` functions are ideal for this.

## Example: JSON Output for Downstream Processing

To store time-delta data in a JSON array for further analysis:

```
SignInLogs
```

```
| where ResultType == "Failed"

| sort by UserPrincipalName, Timestamp asc

| extend TimeDelta = datetime_diff('second',
Timestamp, next(Timestamp, UserPrincipalName))

| where TimeDelta != null

| summarize TimeDeltas = make_list(TimeDelta),
AttemptCount = count() by UserPrincipalName

| extend JsonOutput = toJSON(bag_pack("User",
UserPrincipalName, "Attempts", AttemptCount,
"TimeDeltas", TimeDeltas))

| project JsonOutput
```

## Explanation:

- `make_list(TimeDelta)`: Aggregates time deltas into an array.

- `bag_pack`: Combines user, attempt count, and time deltas into a dynamic object.
- `tojson`: Converts the object to a JSON string, ready for export or processing.

Sample output:

```
{  
  "User": "user@domain.com",  
  "Attempts": 50,  
  "TimeDeltas": [15, 12, 14, 13, ...]  
}
```

This JSON format is easily parsed by downstream systems for alerting, visualization, or further analysis.

## Practical Considerations

- **Performance:** Sorting and `next` operations can be resource-intensive on large datasets. Use `partition by UserPrincipalName` to optimize queries by processing users in parallel.
- **False Positives:** Rapid logins may occur legitimately (e.g., misconfigured apps). Combine time-delta analysis with other signals, like IP addresses or device types, to reduce false positives.
- **Threshold Tuning:** Adjust time-delta thresholds (e.g., 60 seconds) based on your environment's baseline behavior.

KQL's `datetime_diff`, `next`, and statistical functions provide a robust framework for detecting identity attacks like password sprays and brute-force attempts. By calculating inter-event gaps and analyzing patterns, security teams can identify suspicious behavior. Storing results in JSON arrays ensures compatibility with downstream systems, enabling seamless integration into broader security workflows. With these techniques, KQL empowers analysts to stay ahead of evolving threats.

# Chapter 60

---

## External Data Integration and Custom IOC Matching for Enhanced Threat Intelligence

In the world of data analytics and cybersecurity, Kusto Query Language (KQL) stands out as a powerful tool for querying large datasets in platforms like Azure Data Explorer and Microsoft Sentinel. While KQL excels at handling ingested logs and telemetry, integrating external data sources opens up new possibilities for real-time enrichment—particularly in threat hunting scenarios. This chapter dives into two key features: the `externaldata` operator for pulling in data from external storage like CSV or JSON files in Azure Blobs, and dynamic lookups using the `lookup` operator for matching against custom Indicators of Compromise (IOCs). These techniques are especially useful for threat intelligence enrichment but come with challenges around security, data formatting, and performance that make them less commonly explored. We'll cover how they work, provide

practical examples, and discuss the hurdles to watch out for.

## The externaldata Operator: Bringing in External Threat Intel

The externaldata operator allows you to treat external files as temporary tables within your KQL queries, pulling data directly from storage artifacts without needing to ingest it first. This is ideal for reference datasets like threat intelligence feeds, which might include lists of malicious IPs, domains, or file hashes stored in CSV or JSON formats on Azure Blob Storage or Azure Data Lake.

### Syntax and Basics

The basic syntax defines the schema upfront and points to one or more storage URIs:

```
externaldata (columnName:columnType [, ...])  
[storageConnectionString [, ...]] [with  
(propertyName=PropertyValue [, ...])]
```

- **Schema:** Specify columns and types (e.g., ip:string for an IP list).

- **Connection Strings:** Use URIs with authentication like Shared Access Signatures (SAS) or managed identities—avoid embedding credentials directly for security reasons.
- **Properties:** Options like `format="csv"` or `ignoreFirstRecord=true` to skip headers.

Supported formats include CSV, JSON, TXT, Parquet, and more, with automatic detection based on file extensions where possible.

However, it's limited to small datasets (up to 100 MB) and isn't suitable for large-scale ingestion—use custom log ingestion for that instead.

## Example: Enriching Logs with Threat Intel from Blobs

Imagine you have a CSV file in Azure Blob Storage containing known malicious IPs for threat enrichment. Here's a query to match successful sign-ins from those IPs:

```
let BlockList = externaldata(ip:string)
```

[

```
h@“https://storageaccount.blob.core.windows.net  
/container/blocked-ips.csv?SAS_TOKEN”
```

```
]
```

```
with(format=“csv”, ignoreFirstRecord=true)
```

```
| where ip matches regex @“^(25[0-5]|2[0-4][0-  
9]| [01]?[0-9][0-9]?)\\. (25[0-5]|2[0-4][0-  
9]| [01]?[0-9][0-9]?)\\. (25[0-5]|2[0-4][0-  
9]| [01]?[0-9][0-9]?)\\. (25[0-5]|2[0-4][0-  
9]| [01]?[0-9][0-9]?)$”;
```

```
SigninLogs
```

```
| where IPAddress in (BlockList)
```

```
| where ResultType == “0”
```

```
| project Timestamp, UserPrincipalName,  
IPAddress, Location
```

This pulls the IP list dynamically and filters sign-in logs for matches, enabling quick IOC detection.

For JSON, you can use ingestion mappings to parse nested structures:

```
externaldata(Timestamp:datetime, TenantId:guid,  
MethodName:string)
```

```
[
```

```
h@“https://storageaccount.blob.core.windows.net  
/container/events.json?SAS_TOKEN”
```

```
]
```

```
with(format="multijson",  
ingestionMapping='[{"Column":"Timestamp","Properties":{"Path":("$.timestamp")}}, ...]')
```

This is handy for complex threat feeds.

In practice, for threat hunting, you might load domain lists from public GitHub repos or curated blobs to scan inbound emails:

```
let domainList = externaldata(domain:string)
```

```
[@“https://raw.githubusercontent.com/tsirolnik/  
spam-domains-list/master/spamdomains.txt”]
```

```
with(format="txt");
```

```
EmailEvents
```

```
| where EmailDirection == "Inbound"
```

```
| extend EmailDomain =  
tostring(split(SenderMailFromAddress, '@')[1])
```

```
| join kind=inner (domainList) on  
$left.EmailDomain == $right.domain
```

```
| project Timestamp, SenderMailFromAddress,  
EmailDomain, Subject
```

Exclude trusted domains with a datatable to reduce false positives.

## Challenges with externaldata

Security is a big one: Always use read-only SAS tokens or managed identities to avoid exposing sensitive credentials in queries. Format issues arise if the schema doesn't match the file—mismatched columns or invalid data can cause query failures. Dynamic URLs (e.g., date-appended files) require scripting to normalize and upload to stable blobs. Plus, if the storage is firewalled, the operator won't work at all. These factors make it tricky for production use, explaining why it's not as widely adopted.

## Dynamic Lookups: Real-Time Matching Against Watchlists

Once you've loaded external data, the lookup operator shines for efficient, real-time matching. It extends a "fact" table (your main logs) with columns from a smaller "dimension" table (like a watchlist of IOCs), assuming the dimension is compact for performance.

## Syntax and Kinds

```
LeftTable | lookup [kind=(leftouter|inner) ]  
(RightTable) on Attributes
```

- **Kinds:** leftouter (default) includes all left rows with nulls for misses; inner only includes matches.
- **Attributes:** Match on columns, e.g., IPAddress or qualified like \$left.IP == \$right.maliciousIP.

The right table should be under tens of MBs—check with summarize sum(estimate\_data\_size(\*)).

## Example: Matching IOCs with Lookups

Combine with externaldata for dynamic watchlist matching. Load a hash list and lookup against device files:

```
let MaliciousHashes =
externaldata(sha256:string)

[@“https://storageaccount.blob.core.windows.net
/container/emotet-hashes.txt”]

with(format="txt", ignoreFirstRecord=true);

DeviceFileEvents

| lookup kind=inner (MaliciousHashes) on
$left.SHA256 == $right.sha256

| project Timestamp, FileName, SHA256,
DeviceName
```

This enriches file events with matches from an external IOC list in real-time.

For network log enrichment, lookup against Azure IP ranges stored externally:

```
let AzureSubnets = externaldata(Subnet:string,  
Region:string)  
  
[@"https://storageaccount.blob.core.windows.net  
/container/azure-ips.csv?SAS_TOKEN"]  
  
with(format="csv");  
  
AzureNetworkAnalytics_CL  
  
| lookup kind=leftouter (AzureSubnets) on  
$left.DestinationIP == $right.Subnet  
  
| where Region == ""  
  
| project Timestamp, SourceIP, DestinationIP,  
FlowType
```

This tags non-Azure traffic for focused threat analysis.

## **Challenges with Lookups**

Performance drops if the right table is too large, and it's not optimized for massive fact tables—use join for those. Security ties back to the external source, and format mismatches can break the lookup. In threat scenarios, over-reliance on external feeds risks blind spots if they're incomplete.

## **Why Bother with These Techniques?**

Despite the hurdles—credential management, precise formatting, size limits, and the need for scripting to handle dynamic sources—integrating external data via externaldata and lookup unlocks powerful threat intelligence capabilities in KQL. It allows for custom IOC matching without bloating your ingested data, enabling faster detection of anomalies like malicious sign-ins or spam domains. If you're in cybersecurity, start small with public feeds and secure blobs to experiment. With careful setup, these tools can significantly boost your threat hunting game in Microsoft Sentinel or Azure Data Explorer.

# Chapter 61

---

## Advanced KQL for Logic Apps Monitoring: Harnessing Custom Dimensions in Application Insights

In the world of serverless orchestration, Azure Logic Apps stand out for their ability to automate workflows with minimal overhead. But as these workflows grow in complexity—integrating APIs, handling high volumes of triggers, and spanning multiple actions—monitoring becomes crucial.

Enter Application Insights, Azure's powerhouse for telemetry, where logs are rich but often buried in JSON payloads. While Azure documentation skims the surface, advanced parsing of these logs using Kusto Query Language (KQL) remains under-explored outside the Azure echo chamber.

This post dives into practical KQL techniques for Logic Apps Standard (the single-tenant flavor), focusing on two high-impact areas: extracting custom dimensions for run IDs and action durations to spotlight the slowest performers, and aggregating API call metrics by URL while

filtering out noisy internal hooks. These queries, drawn from real-world patterns, will help you shift from reactive firefighting to proactive optimization. Whether you're troubleshooting latency spikes or auditing external dependencies, KQL's flexibility shines here.

We'll use the traces and requests tables in Application Insights, where Logic Apps telemetry lands. Custom dimensions—those extensible key-value pairs in telemetry events—are your gateway to granular insights. Always enable diagnostic settings to route Logic Apps logs to App Insights, and consider debug-level verbosity for deeper traces (just watch your ingestion costs).

## **Parsing Custom Dimensions: Unlocking Run IDs and Durations**

Logic Apps actions emit traces with embedded JSON in customDimensions, holding gems like run IDs, workflow names, and execution times. Parsing this requires `parse_json()` to navigate the nested structure, often under `prop__properties` or `resource`.

Here's a foundational query to extract run IDs, workflow details, and durations from traces. It

**filters for action-level events and projects key fields for easy scanning:**

traces

```
| where cloud_RoleName contains "logicapp" //  
Filter to Logic Apps instances
```

```
| where timestamp > ago(7d) // Last 7 days
```

```
| extend properties =  
parse_json(tostring(customDimensions.prop__prop  
erties))
```

```
| extend resource = properties.resource
```

```
| extend actionDetails =  
parse_json(tostring(customDimensions.prop__acti  
on))
```

```
| project
```

```
timestamp,  
  
WorkflowName =  
tostring(resource.workflowName),  
  
RunId =  
tostring(customDimensions.prop__flowRunSequence  
Id),  
  
ActionName = tostring(actionDetails.name),  
  
Status =  
tostring(customDimensions.prop__status),  
  
DurationMs = toint(actionDetails.endTime) -  
toint(actionDetails.startTime) // Milliseconds  
  
| where isnotempty(RunId) and  
isnotempty(ActionName)  
  
| order by timestamp desc
```

This query reveals the “who, what, and how long” of each action. The DurationMs calculation derives from start/end timestamps in the action details—crucial since raw traces don’t always expose a top-level duration. Run it in the App Insights query editor to drill into specific runs via the extracted RunId, which you can hyperlink back to the Logic Apps designer.

## Top-N Summaries: Hunting the Slowest Actions

With extracted durations in hand, summarizing for outliers is straightforward. This next query builds on the above, aggregating average durations by action and surfacing the top 10 slowest across successful runs. It excludes failures to focus on performance bottlenecks, not errors:

traces

```
| where cloud_RoleName contains "logicapp" and  
timestamp > ago(7d)
```

```
| where customDimensions.prop__status ==  
"Succeeded" // Successful actions only  
  
| extend properties =  
parse_json(tostring(customDimensions.prop__prop  
erties))  
  
| extend resource = properties.resource  
  
| extend actionDetails =  
parse_json(tostring(customDimensions.prop__acti  
on))  
  
| extend DurationMs =  
toint(actionDetails.endTime) -  
toint(actionDetails.startTime)  
  
| where isnotempty(DurationMs) and DurationMs >  
0  
  
| summarize
```

```
AvgDurationMs = avg(DurationMs),  
  
TotalRuns = count(),  
  
MinDurationMs = min(DurationMs),  
  
MaxDurationMs = max(DurationMs)  
  
by WorkflowName =  
tostring(resource.workflowName), ActionName =  
tostring(actionDetails.name)  
  
| top 10 by AvgDurationMs desc  
  
| project WorkflowName, ActionName,  
AvgDurationMs, TotalRuns, MinDurationMs,  
MaxDurationMs  
  
| extend AvgDurationSec = AvgDurationMs /  
1000.0
```

Output might look like this (hypothetical results):

WorkflowName	ActionName	AvgDurationSec	TotalRuns	MinDurationMs	MaxDurationMs	⋮
OrderProcessor	HTTP-PostToCRM	4.2	150	500	12000	
InventorySync	ParseJSON-Data	2.8	200	200	8000	
NotificationFlow	SendEmail	1.5	300	100	5000	

Spotting a CRM post averaging 4+ seconds? Time to investigate throttling or payload bloat. Tweak the top clause for your N, or add | render barchart for visuals.

## Aggregating API Call Metrics: Focus on External Dependencies

HTTP actions in Logic Apps often manifest as requests in App Insights, capturing URL, duration, and success metrics. But internal callbacks (e.g., webhook endpoints like /runtime/webhooks) clutter the view. This aggregation query excludes them, rolling up total, average, and max durations by URL—ideal for dependency SLAs.

requests

```
| where cloud_RoleName contains "logicapp" and
| timestamp > ago(7d)

| where success == true // Successful calls
only

| where isnotempty(url) and not(url contains
"runtime/webhooks") // Exclude internal hooks

| extend DurationSec = duration / 1000.0

| summarize

TotalDurationSec = sum(DurationSec) ,

AvgDurationSec = avg(DurationSec) ,

MaxDurationSec = max(DurationSec) ,
```

```
CallCount = count()  
  
by url  
  
| order by MaxDurationSec desc  
  
| top 10 by MaxDurationSec  
  
| project url, AvgDurationSec, MaxDurationSec,  
TotalDurationSec, CallCount
```

## Sample results:

url	AvgDurationSec	MaxDurationSec	TotalDurationSec	CallCount	⋮
<a href="https://api.externalcrm.com/v1/orders">https://api.externalcrm.com/v1/orders</a>	1.8	15.2	450.0	250	🔗
<a href="https://graph.microsoft.com/v1.0/notifications">https://graph.microsoft.com/v1.0/notifications</a>	0.9	8.5	225.0	250	🔗
<a href="https://storage.blob.core.windows.net/">https://storage.blob.core.windows.net/...</a>	0.4	3.1	100.0	250	🔗

The max outlier screams for a retry policy or async pattern. Group further by workflow with by url, cloud\_RoleName if needed, or join back to traces for run context.

# Elevate Your Logic Apps Observability

These KQL patterns transform raw App Insights logs into actionable intelligence, far beyond Azure's canned dashboards. Start simple—paste these into the query editor—and iterate with filters for your environment. Pro tips: Use union traces, requests for holistic views, set up alerts on query results via Azure Monitor, and export to Power BI for stakeholder reports.

Monitoring isn't just about logs; it's about foresight. By mastering custom dimensions and targeted aggregations, you'll keep your Logic Apps humming efficiently, even as they scale.

# Chapter 62

---

## The KQL Cheat Sheet

**Download:** <https://github.com/rod-trent/MustLearnKQL/blob/main/Docs/Kusto%20Query%20Language%20Cheat%20Sheet.pdf>

This chapter introduces a concise **Kusto Query Language (KQL)** cheat sheet for Azure Data Explorer, designed as a quick reference for querying large datasets. Each item in the cheat sheet is explained below with practical examples to help you write effective KQL queries. Download the cheat sheet above for easy access.

## Operators

### 1. where

- Filters rows based on a condition, narrowing down results.
- **Example:** `StormEvents | where DamageProperty > 10000` returns events with property damage exceeding \$10,000.

## 2. project

- Selects specific columns to include in the output, reducing data clutter.
- **Example:** Logs | project Timestamp, User outputs only the Timestamp and User columns from the Logs table.

## 3. join

- Combines two tables based on a common key, with options like inner, leftouter, etc.
- **Example:** Requests | join kind=inner Logs on RequestId merges Requests and Logs tables where RequestId matches.

## 4. union

- Combines rows from multiple tables or queries into a single result set.
- **Example:** union Errors, Warnings combines rows from Errors and Warnings tables.

## 5. search

- Searches for a term across all columns, ideal for exploratory queries.

- **Example:** Logs | search "error" finds "error" in any column of the Logs table.

## 6. == (Equality)

- Tests if a column exactly matches a value.
- **Example:** Users | where Name == "john" returns rows where the Name column is exactly "john".

## 7. != (Inequality)

- Filters rows where a column does not match a value.
- **Example:** Logs | where Status != 200 excludes successful HTTP responses.

## 8. in

- Checks if a column value is in a specified set, simplifying multiple comparisons.
- **Example:** Logs | where Status in (400, 404, 500) matches bad request, not found, or server error status codes.

## 9. has

- Matches rows where a column contains a term (case-sensitive with has\_cs).

- **Example:** Logs | where Message has "error" finds rows with "error" in the Message column.

## Functions

### 1. `count()`

- Counts rows in a group or result set, useful for summarizing data.
- **Example:** Logs | summarize count() by Status counts occurrences of each HTTP status code.

### 2. `avg()`

- Calculates the average of a numeric column.
- **Example:** Requests | summarize avg(Duration) computes the average request duration.

### 3. `sum()`

- Sums values in a numeric column.
- **Example:** Logs | summarize sum(Bytes) totals the bytes transferred in the Logs table.

### 4. `max()`

- Finds the highest value in a column.

- **Example:** Events | summarize max(Duration) identifies the longest event duration.

## 5. min()

- Finds the lowest value in a column.
- **Example:** Requests | summarize min(Latency) returns the shortest latency.

## 6. ago()

- Specifies a time relative to the current time for filtering.
- **Example:** Logs | where Timestamp > ago(1h) filters logs from the last hour.

## 7. tostring()

- Converts a value to a string for display or comparison.
- **Example:** Logs | project tostring(Status) converts the Status column to strings.

## Syntax

### 1. | (Pipe)

- Chains operations, passing results from one step to the next.

- **Example:** Logs | where Status == 200 | project User filters for successful responses and selects the User column.

## 2. **summarize**

- Groups data and applies aggregations like count or sum.
- **Example:** Logs | summarize count() by User counts log entries per user.

## 3. **order by**

- Sorts results by a column in ascending (asc) or descending (desc) order.
- **Example:** Logs | order by Timestamp desc sorts logs by timestamp, newest first.

## 4. **take**

- Limits the number of rows returned, useful for sampling.
- **Example:** Logs | take 100 returns the first 100 rows of the Logs table.

## 5. **extend**

- Adds calculated columns to the result set.
- **Example:** Requests | extend ResponseTimeSec = Duration /

1000 creates a new column with duration in seconds.

## 6. Range Queries

- Filters values within a range using between or comparison operators.
- **Example:** Logs | where Status between(400 .. 499) matches client error status codes (400–499).

## 7. Case Sensitivity

- KQL operators are case-insensitive, but string functions like has have case-sensitive variants (has\_cs).
- **Example:** Logs | where Message has "ERROR" may match "error" unless using has\_cs.

## 8. Comments

- Single-line comments start with //, ignored during query execution.
- **Example:** Logs | where Status == 200 // Filter successful requests includes a comment for clarity.

This KQL cheat sheet and its explanations provide a quick reference for querying data in Azure Data Explorer. Download the cheat sheet

above and keep it handy for your next KQL session!

# Chapter 63

---

## Who does it better? Or Worse? Generative AI for KQL

I had an upcoming session where I would be talking about the various Generative AI infused tools to help security analysts and engineers. I intended to show demos of the differences between the tools and making recommendations on which to use in specific scenarios.

To get started preparing, I started with a simple prompt to get a very specific KQL query (or queries). Here you can see what I saw and check out the differences.

This topic comes up quite a bit these days. Many people feel that ChatGPT and other Generative AI tools should be able to produce quality KQL queries. In a lot of respects, it can – but there are gotchas. On one hand, these tools know nothing about your own environment – meaning they don't know the tables and data you have to query against. Secondly, the queries they produce don't always pay attention to best practices so you

have to still know KQL to tweak them for performance and accuracy. All that said, some of the existing tools are better than the others. Here's my foray into determining which is best for my purposes. Your experiences may vary.

## The Prompt

*How can I use a KQL to query to identify the following suspicious behavior?*

*Unusual login attempts or failed logins across devices and patterns that indicate potential data exfiltration or unauthorized file access.*

## Copilot

To identify suspicious behaviors like unusual login attempts, failed logins, and patterns of data exfiltration using KQL, you can construct queries that analyze and filter logs for specific indicators. Here's how you can approach each case:

### **1. Unusual Login Attempts or Failed Logins Across Devices**

- Focus on event logs that capture authentication activities, such as `SigninLogs` or similar datasets.
- Use filters for failed logins (`resultType: "Failure"`) and group data by user accounts or devices to spot anomalies.

Here's an example query:

```
• SigninLogs  
  
• | where ResultType == "Failure"  
  
• | summarize FailureCount = count() by UserPrincipalName, DeviceDisplayName,  
bin(TimeGenerated, 1h)  
  
• | where FailureCount > 5  
  
• | order by FailureCount desc
```

This identifies users or devices with multiple failed login attempts within a short timeframe.

## 2. Patterns Indicating Potential Data Exfiltration or Unauthorized File Access

- Use logs that track file activities or access attempts, like `FileAuditLogs` or endpoint activity logs.
- Look for excessive file access or unusual data transfer patterns (e.g., large data volumes, rare access times, or specific file types).

Here's an example query:

```
• FileAuditLogs  
  
• | where ActionType == "FileAccess"  
  
• | summarize FileAccessCount = count(),  
DataVolume = sum(FileSize) by  
UserPrincipalName, bin(TimeGenerated, 1h)  
  
• | where FileAccessCount > 10 or  
DataVolume > 10485760 // Thresholds for  
anomaly detection
```

- | order by DataVolume desc

This flags users with unusually high file access or data transfer activity.

## Enhancements

For both scenarios, you can:

- Correlate data across multiple logs (e.g., authentication and file access logs).
- Add IP address and geolocation filters to detect activity from uncommon or flagged locations.
- Include time-based comparisons to identify deviations from typical behavior.

Would you like to refine these queries further based on specific thresholds or datasets you're working with? Let's elevate this investigation together!

## Grok

To identify suspicious behavior like unusual login attempts, failed logins, or patterns indicating potential data exfiltration or unauthorized file access using KQL (Kusto Query

Language), you can craft queries targeting relevant data sources such as Azure Active Directory (AAD) logs, Microsoft 365 audit logs, or endpoint security logs (e.g., Microsoft Defender for Endpoint). Below are example KQL queries tailored to these scenarios, assuming you're querying in a Microsoft Sentinel or Azure Data Explorer environment.

---

## 1. Unusual Login Attempts or Failed Logins

This query looks for failed login attempts, multiple failed logins from the same user across devices, or logins from unusual locations/IPs.

- SigninLogs
- | where TimeGenerated > ago(7d)
- | where ResultType != 0 // Non-successful logins (failed attempts)

- | summarize FailedLoginCount = count(), DistinctDevices = dcount(DeviceDetail.deviceId), DistinctIPs = dcount(IPAddress), Locations = make\_set(Location)
- by UserPrincipalName, AppDisplayName
- | where FailedLoginCount > 5 or DistinctIPs > 2 or array\_length(Locations) > 2
- | project UserPrincipalName, FailedLoginCount, DistinctDevices, DistinctIPs, Locations, AppDisplayName
- | order by FailedLoginCount desc

## Explanation:

- SigninLogs: Contains Azure AD sign-in events.
- ResultType != 0: Filters for failed logins (success is 0).

- summarize: Aggregates failed login counts, devices, IPs, and locations per user.
  - where FailedLoginCount > 5 or DistinctIPs > 2 or array\_length(Locations) > 2: Flags users with high failure rates, multiple IPs, or diverse locations.
  - Adjust thresholds (5, 2) based on your environment.
- 

## 2. Suspicious Patterns Across Devices

This query detects users logging in from multiple devices in a short time frame, which could indicate compromised credentials.

- SigninLogs
- | where TimeGenerated > ago(1d)
- | where ResultType == 0 // Successful logins

```
• | summarize DeviceCount =  
dcount(DeviceDetail.deviceId), Devices =  
make_set(DeviceDetail.displayName)  
  
• by UserPrincipalName,  
bin(TimeGenerated, 1h)  
  
• | where DeviceCount > 2  
  
• | project TimeGenerated,  
UserPrincipalName, DeviceCount, Devices  
  
• | order by DeviceCount desc
```

## Explanation:

- Filters for successful logins (ResultType == 0).
- bin(TimeGenerated, 1h): Groups events into 1-hour windows.
- dcount(DeviceDetail.deviceId): Counts unique devices per user.

- where DeviceCount > 2: Flags users accessing from more than 2 devices in an hour (adjust as needed).
- 

### 3. Potential Data Exfiltration

This query looks for patterns like large file downloads, unusual file sharing, or access to sensitive files, using audit logs (e.g., Office 365 or SharePoint activity).

- OfficeActivity
- | where TimeGenerated > ago(7d)
- | where Operation in ("FileDownloaded", "FileSyncDownloadedFull", "FileShared", "FileAccessed")
- | summarize FileCount = count(), TotalBytes = sum(if(isnotempty(FileSize),

```
tolong(FileSize), 0)), DistinctFiles =  
dcount(SourceFileName)  
  
• by UserId, ClientIP, Operation  
  
• | where FileCount > 50 or TotalBytes >  
100000000 // Approx 100MB  
  
• | project UserId, ClientIP, FileCount,  
TotalBytes, DistinctFiles, Operation  
  
• | order by TotalBytes desc
```

## Explanation:

- OfficeActivity: Contains file-related activities in Office 365 (SharePoint, OneDrive, etc.).
- Operation in (...): Targets file download, sync, sharing, or access events.
- sum(if(...)): Aggregates total bytes downloaded (handles missing FileSize).
- where FileCount > 50 or TotalBytes > 100000000: Flags users downloading

many files or large data volumes (adjust thresholds).

---

## 4. Unauthorized File Access

This query detects repeated failed attempts to access files or access to sensitive files by users without prior history.

- OfficeActivity
  
  
  
  
  
- | where TimeGenerated > ago(7d)
  
  
  
  
  
- | where Operation == "FileAccessFailed"  
or (Operation == "FileAccessed" and  
SourceFileName contains "sensitive")
  
  
  
  
  
- | summarize AccessAttempts = count(),  
DistinctFiles = dcount(SourceFileName)
  
  
  
  
  
- by UserId, Operation, ClientIP

- | where AccessAttempts > 10
- | project UserId, Operation, AccessAttempts, DistinctFiles, ClientIP
- | order by AccessAttempts desc

## **Explanation:**

- Operation == "FileAccessFailed": Captures failed file access attempts.
  - SourceFileName contains "sensitive": Flags access to files with sensitive keywords (customize as needed).
  - where AccessAttempts > 10: Highlights users with excessive failed attempts.
- 

## **5. Combining Indicators for Suspicious Behavior**

To correlate failed logins with potential exfiltration, you can join SigninLogs and OfficeActivity:

- let SuspiciousUsers = SigninLogs
  - | where TimeGenerated > ago(1d)
- | where ResultType != 0
- | summarize FailedLogins = count() by UserPrincipalName
- | where FailedLogins > 5;
- OfficeActivity
  - | where TimeGenerated > ago(1d)
- | where Operation in ("FileDownloaded", "FileShared")

- | join kind=inner SuspiciousUsers on \$left.UserId == \$right.UserPrincipalName
- | summarize FileCount = count(), TotalBytes = sum(iff(isnotempty(FileSize), tolong(FileSize), 0))
- by UserId, ClientIP
- | where FileCount > 20 or TotalBytes > 50000000
- | project UserId, ClientIP, FileCount, TotalBytes

## Explanation:

- SuspiciousUsers: Identifies users with many failed logins.
- join: Correlates with file activity from those users.
- Flags users who failed logins and then downloaded/shared files.

---

## Tips for Refining Queries

- **Adjust Time Windows:** Use ago(1h), ago(1d), or ago(7d) based on your monitoring needs.
- **Tune Thresholds:** Modify counts (> 5, > 50) to reduce false positives.
- **Filter by Environment:** Add conditions like AppDisplayName == "SharePoint Online" or DeviceDetail.operatingSystem to focus on specific platforms.
- **Enrich with Context:** Include UserAgent, Location, or RiskLevel from SigninLogs for deeper analysis.
- **Schedule Alerts:** Use Microsoft Sentinel to run these queries periodically and trigger alerts.

If you have specific data sources or scenarios in mind, let me know, and I can tailor the queries further!

## ChatGPT

To identify unusual login attempts, failed logins across multiple devices, and patterns that suggest data exfiltration or unauthorized

**access using Kusto Query Language (KQL) in environments like Microsoft Sentinel or Defender, you'd typically correlate logs from **SigninLogs**, **AuditLogs**, and **DeviceFileEvents** or **DeviceNetworkEvents** (if Defender for Endpoint is enabled).**

Here's a breakdown of how to approach each indicator and sample queries for them:

---

## 1. Unusual Login Attempts

Detect sign-ins from **new countries, rare devices, or impossible travel scenarios**.

- SigninLogs
- | extend DeviceDetail =  
tostring(DeviceDetail.deviceDisplayName)
- | extend Location =  
tostring(LocationDetails.countryOrRegion)

- | summarize Count = count(),
- Devices = dcount(DeviceDetail),
- Locations = dcount(Location)
- by UserPrincipalName,  
bin(TimeGenerated, 1d)
- | where Devices > 3 or Locations > 2

Optional: Cross-check against previous behavior to detect anomalies.

---

## 🚫 2. Multiple Failed Login Attempts Across Devices

Helps detect potential brute-force or credential stuffing attempts.

- SigninLogs
- | where ResultType != 0 // 0 = success
- | summarize FailedAttempts = count(),  
UniqueDevices =  
dcount(DeviceDetail.deviceDisplayName)
- by UserPrincipalName,  
bin(TimeGenerated, 1h)
- | where FailedAttempts >= 5 and  
UniqueDevices >= 3

---

## 3. Potential Data Exfiltration

Look for large outbound file transfers or access to many files in a short time (if Defender or Purview logs are available).

## **Option A: DeviceFileEvents (Defender for Endpoint)**

- DeviceFileEvents
- | where ActionType == "FileCopied" or ActionType == "FileMoved"
- | where FileName endswith ".docx" or FileName endswith ".xlsx" or FileName endswith ".pdf"
- | summarize FileAccesses = count(), UniqueDestinations = dcount(RemoteDeviceName)
- by DeviceName,  
InitiatingProcessAccountName, bin(Timestamp, 1h)
- | where FileAccesses > 50

## **Option B: SharePoint or OneDrive access via AuditLogs**

- AuditLogs
- | where Operation in ("FileAccessed", "FileDownloaded", "FileViewed", "FileAccessedExtended")
- | where Site\_Url has\_any ("sharepoint.com", "onedrive.com")
- | summarize AccessCount = count(), UniqueFiles = dcount(ObjectId)
- by UserId, bin(TimeGenerated, 1h)
- | where AccessCount > 100 or UniqueFiles > 50

---

## 4. Correlating Logins with File Access

Suspicious behavior might involve a failed login flood followed by successful login and file access.

- ```
let FailedLogins = SigninLogs
```
- ```
| where ResultType != 0
```
- ```
| summarize FailedAttempts = count() by UserPrincipalName, bin(TimeGenerated, 1h);
```
- 
- ```
let FileAccess = DeviceFileEvents
```
- ```
| where ActionType == "FileRead"
```
- ```
| summarize FileReads = count() by InitiatingProcessAccountName, bin(Timestamp, 1h);
```

- - FailedLogins
  - | join kind=inner (
  - | project UserPrincipalName = InitiatingProcessAccountName, FileReads, Timestamp
  - ) on UserPrincipalName
  - | where FileReads > 50 and FailedAttempts > 5
-

## Tips

- Replace `DeviceFileEvents` and `DeviceNetworkEvents` with actual data sources available in your tenant (e.g., `OfficeActivity` for SharePoint/OneDrive).
  - Add user risk signals from **IdentityInfo** or Microsoft Defender XDR if enabled.
  - Use `join` operations to correlate across time and behavior.
- 

Let me know your exact data sources (Sentinel, Defender for Endpoint, Purview, etc.), and I can tailor these further.

# Microsoft Security Copilot

How can I use KQL to query to identifying the following suspicious behavior? Unusual login attempts or failed logins across devices and patterns that indicate potential data exfiltration or unauthorized file access

- ✓ 3 steps completed 1 minute 14 seconds
  - > Chose Natural language to KQL for advanced hunting 6 seconds
  - > Generated KQL query 23 seconds
  - > Prepared your response 45 seconds

Apr 15, 4:41 PM

#### Kusto Query to Identify Suspicious Behavior

```
1 // Identify Unusual Login Attempts or Failed Logins
2 let UnusualLogins = DeviceLogonEvents
3 | where ActionType in ("LogonFailed", "LogonAttempted")
4 | project TimeGenerated, DeviceName, AccountName, RemoteIP, ActionType,
  LogonType
5 // Identify Potential Data Exfiltration or Unauthorized File Access
6 let UnauthorizedFileAccess = DeviceFileEvents
7 | where ActionType in ("FileCreated", "FileDeleted", "FileModified",
  "FileRead")
8 | project TimeGenerated, DeviceName, RequestAccountName, FileName,
  FolderPath, ActionType;
9 | order by TimeGenerated desc
```

#### Explanation of the Kusto Query

The Kusto query is designed to identify unusual login attempts or failed logins, as well as potential data exfiltration or unauthorized file access. It combines results from two different data sources: `DeviceLogonEvents` and `DeviceFileEvents`, and orders the combined results by the time they were generated.

#### Breakdown of the Kusto Query

- // Identify Unusual Login Attempts or Failed Logins :Comment indicating the purpose of the following section.
- let UnusualLogins = DeviceLogonEvents :Defines a variable `UnusualLogins` to store filtered login events.
- | where ActionType in ("LogonFailed", "LogonAttempted") :Filters the login events to include only failed or attempted logins.
- | project TimeGenerated, DeviceName, AccountName, RemoteIP, ActionType, LogonType :Selects specific columns to include in the output for the login events.
- // Identify Potential Data Exfiltration or Unauthorized File Access :Comment indicating the purpose of the following section.
- let UnauthorizedFileAccess = DeviceFileEvents :Defines a variable `UnauthorizedFileAccess` to store filtered file events.
- | where ActionType in ("FileCreated", "FileDeleted", "FileModified", "FileRead") :Filters the file events to include only those indicating potential unauthorized access.
- | project TimeGenerated, DeviceName, RequestAccountName, FileName, FolderPath, ActionType :Selects specific columns to include in the output for the file events.
- // Combine Results :Comment indicating the purpose of the following section.
- UnusualLogins :References the `UnusualLogins` variable.
- | union UnauthorizedFileAccess :Combines the results from `UnusualLogins` and `UnauthorizedFileAccess`.
- | order by TimeGenerated desc :Orders the combined results by the time they were generated, in descending order.

#### Query Results

Export to Excel

TimeGenerated	DeviceName	AccountName
2025-04-15T20:41:00.040285Z	vnevado-linux	
2025-04-15T20:40:34.503356Z	vnevado-linux	
2025-04-15T20:40:33.087392Z	vnevado-linux	
2025-04-15T20:40:32.532067Z	vnevado-linux	

ⓘ How's this response?

# Microsoft 365 Copilot

Certainly! Here are some KQL queries to help you identify suspicious behavior:

## Unusual Login Attempts

To detect unusual login attempts, you can use the following query to identify multiple failed login attempts within a short time frame:

- ```
let failed_threshold = 5; // threshold for failed login times
```
- ```
let failed_count = 2; // threshold for number of failed logins
```
- ```
let start_time = startofday(datetime(2025-04-01)); // start date
```
- ```
let end_time = endofday(datetime(2025-04-15)); // end date
```

- SecurityEvent
- | where TimeGenerated between (start\_time .. end\_time)
- | where EventID == 4625 // Event ID for failed login
- | project Account, TimeGenerated, Computer
- | sort by TimeGenerated asc, Account
- | serialize
- | extend nextAccount = next(Account, 1), nextTimeGenerated = next(TimeGenerated, 1)
- | where Account == nextAccount

- | extend TimeDeltaInSeconds =  
datetime\_diff("second", nextTimeGenerated,  
TimeGenerated)
- | where TimeDeltaInSeconds <=  
failed\_threshold
- | project TimeGenerated,  
TimeDeltaInSeconds, Account, Computer
- | summarize Failed\_Logins = count(),  
avg(TimeDeltaInSeconds), first\_failed =  
min(TimeGenerated), last\_failed =  
max(TimeGenerated), standarddev =  
stdev(TimeDeltaInSeconds), variance =  
variance(TimeDeltaInSeconds), TimeDeltaList =  
make\_list(TimeDeltaInSeconds) by Account
- | where standarddev < 1
- | where Failed\_Logins >= failed\_count
- | sort by Failed\_Logins desc

## Data Exfiltration

To detect potential data exfiltration, you can use the following query to identify large file transfers:

- let TermAccount = 'departing.employee';  
// Enter the departing employee's username
- let ReleaseTime = datetime("2025-04-01 00:00:00"); // Enter the date the resignation or termination was announced
- DeviceNetworkEvents
- | where InitiatingProcessAccountName =~ TermAccount
- | where Timestamp > ReleaseTime
- | join kind=inner (DeviceFileEvents) on InitiatingProcessAccountName

- | where FileName endswith ".docx" or  
FileName endswith ".pptx" or FileName endswith  
".xlsx" or FileName endswith ".pdf"
- | join DeviceNetworkInfo on DeviceId
- | where ConnectedNetworks !contains  
'\"Category\":\"Domain\"' // Looking for  
remote, non-domain networks
- | summarize TotalFiles = count() by  
bin(Timestamp, 5m),  
InitiatingProcessAccountName
- | where TotalFiles > 1000 // adjust  
accordingly
- | project TotalFiles, Timestamp,  
InitiatingProcessAccountName

## Unauthorized File Access

To detect unauthorized file access, you can use the following query:

- DeviceFileEvents
- | where ActionType == "FileAccessed"
- | where FileName endswith ".docx" or  
FileName endswith ".pptx" or FileName endswith  
".xlsx" or FileName endswith ".pdf"
- | where AccountName != "authorized\_user"  
// Replace with the authorized user's account  
name
- | summarize FileAccessCount = count() by  
AccountName, FileName
- | where FileAccessCount > 10 // adjust  
threshold as needed

- | project AccountName, FileName,  
FileAccessCount

These queries should help you monitor and identify suspicious activities related to login attempts, data exfiltration, and unauthorized file access[\[1\]](#) [\[2\]](#) [\[3\]](#) [\[4\]](#).

Do you need any further assistance with these queries?

## References

- [1] [function - Assistance with Kusto Query Language \(KQL\) for Identifying ...](#)
- [2] [KQL Query for failed logins · GitHub](#)
- [3] [KQL-Queries/FailedLoginAttempts.kql at Main · EEN421/KQL-Queries - GitHub](#)
- [4] [KQL-Scripts/Detect unauthorized access attempts to Azure ... - GitHub](#)

## Gemini

You can use Kusto Query Language (KQL) to identify unusual login attempts, failed logins across devices, and patterns indicative of

potential data exfiltration or unauthorized file access by analyzing various security logs within Microsoft Defender for Cloud, Microsoft Sentinel, and other relevant data sources.

Here's a breakdown of how you can construct KQL queries for each of these scenarios, along with explanations and considerations:

## **1. Unusual Login Attempts and Failed Logins Across Devices:**

This involves looking for login patterns that deviate from a user's normal behavior, such as logins from new or multiple unusual locations, at unusual times, or a high number of failed attempts followed by a successful login.

### **Code snippet**

- // Look for successful logins from new or unusual locations for a user
- SigninLogs

- | where UserId == "<UserPrincipalName>"  
// Replace with the specific user or remove for all users
- | summarize arg\_min(TimeGenerated, Location) by UserId, Location
- | where count() > <Threshold> // Set a threshold for the number of distinct locations
- | extend UnusualLocation = iff(Location !in (<FrequentLocations>), "Unusual", "Frequent") // Optional: Compare against known frequent locations
- | where UnusualLocation == "Unusual"
- | project TimeGenerated, UserId, Location, IPAddress, UserAgent
-

- // Look for a high number of failed logins followed by a successful login for a user
- SigninLogs
- | where UserId == "<UserPrincipalName>"  
// Replace with the specific user or remove for all users
- | make-series FailedCount = countif(ResultType != 0) default=0 on TimeGenerated in range(ago(1d), now(), 1h)
- | mvexpand TimeGenerated, FailedCount
- | where FailedCount > <FailedLoginThreshold> // Set a threshold for the number of failed attempts in an hour
- | join kind=inner (

- SigninLogs
- | where UserId == "" and ResultType == 0 // Successful logins
- | project SuccessfulLoginTime = TimeGenerated
- ) on \$left.TimeGenerated == \$right.SuccessfulLoginTime
- | project FailedLoginTime = TimeGenerated, UserId, IPAddress, UserAgent, SuccessfulLoginTime
- 
- // Look for logins from multiple devices within a short timeframe for a user

- SigninLogs
- | where UserId == "<UserPrincipalName>"  
// Replace with the specific user or remove for all users
- | summarize dcount(DeviceId) by UserId,  
bin(TimeGenerated, 1h)
- | where dcount\_DeviceId >  
<MultipleDeviceThreshold> // Set a threshold for the number of devices in an hour
- | join kind=inner (
- SigninLogs
- | where UserId ==  
"<UserPrincipalName>"

- | project TimeGenerated, UserId, DeviceId, IPAddress, UserAgent, Location
- ) on \$left.UserId == \$right.UserId and \$left.TimeGenerated == bin(\$right.TimeGenerated, 1h)
- | project TimeGenerated, UserId, DeviceId, IPAddress, UserAgent, Location, DeviceCount = dcount\_DeviceId

## Key Considerations:

- **SigninLogs table:** This table in Azure Monitor/Sentinel contains information about Azure Active Directory sign-ins.
- **<UserPrincipalName>:** Replace this with a specific user if you are investigating a particular account.
- **<Threshold>:** Define a reasonable threshold for the number of distinct locations to consider unusual. This will depend on your environment.
- **<FrequentLocations>:** (Optional) Replace this with a comma-separated list of locations that are considered normal for your users.

- <FailedLoginThreshold>: Set a threshold for the number of failed login attempts within a specific timeframe (e.g., per hour) that you consider suspicious.
- <MultipleDeviceThreshold>: Set a threshold for the number of distinct devices a user logs in from within a short timeframe.
- **Time Window:** Adjust the time range (`ago(1d), 1h`) as needed for your analysis.
- **Baselines:** Establishing baselines for user behavior (e.g., typical login locations, devices) can significantly improve the accuracy of these queries. You might need to create separate queries to build these baselines over time.
- **Noise Reduction:** Expect some false positives. You'll need to fine-tune your thresholds and potentially incorporate more contextual information to reduce noise.

## 2. Patterns Indicating Potential Data Exfiltration:

Identifying data exfiltration through logs can be challenging as it often involves legitimate actions performed in a malicious way. However, you can look for patterns that might suggest unusual data access or transfer.

## Code snippet

- // Look for a high volume of file downloads by a single user within a short timeframe
- AuditLogs
- | where OperationName contains "Download" and UserId == "<UserPrincipalName>" // Adjust OperationName based on your data source
- | summarize DownloadCount = count() by UserId, bin(TimeGenerated, 1h)
- | where DownloadCount > <HighDownloadThreshold> // Set a threshold for the number of downloads per hour
- | project TimeGenerated, UserId, DownloadCount

- - // Look for access to an unusually large number of files or folders by a user
- StorageFileLogs // Replace with the relevant table for file access logs
- | where UserId == "<UserPrincipalName>" and AccessType in ("Read", "List") // Adjust AccessType as needed
- | summarize AccessedResourceCount = dcount(ObjectName) by UserId, bin(TimeGenerated, 1d)
- | where AccessedResourceCount > <HighResourceAccessThreshold> // Set a threshold for the number of unique resources accessed per day
- | project TimeGenerated, UserId, AccessedResourceCount

- 
- // Look for external sharing of files or folders (if your logs capture this)
- AuditLogs
- | where OperationName contains "Share" and SharingType == "External" and UserId == "<UserPrincipalName>" // Adjust OperationName and SharingType
- | project TimeGenerated, UserId, TargetResource, SharedWith
- 
- // Look for unusual network traffic patterns (requires Network Security Group logs or similar)

- AzureNetworkAnalytics\_CL
- | where SourceObjectId == "<UserObjectId>" or DestinationObjectId == "<UserObjectId>" // Filter by user's associated objects
- | summarize TotalBytesTransferred = sum(TotalBytesTransferred) by SourceIP, DestinationIP, DestinationPort, Protocol
- | order by TotalBytesTransferred desc
- | take <TopN> // Look for unusually high traffic volumes to specific external IPs or ports
- // Look for multiple successful logins followed by a large number of file access/downloads

- let suspiciousLogins = SigninLogs
- | where UserId == "<UserPrincipalName>"  
and ResultType == 0
- | project SuccessfulLoginTime =  
TimeGenerated;
- StorageFileLogs // Or relevant file  
access logs
- | where UserId == "<UserPrincipalName>"  
and AccessType in ("Read", "Download")
- | join kind=inner suspiciousLogins on  
\$left.TimeGenerated >  
\$right.SuccessfulLoginTime and  
\$left.TimeGenerated <  
\$right.SuccessfulLoginTime + 1h // Look for  
activity within 1 hour of login
- | summarize FileActionCount = count() by  
UserId, bin(TimeGenerated, 1h)

- | where FileActionCount > <HighFileActionThresholdAfterLogin>
- | project TimeGenerated, UserId, FileActionCount

## Key Considerations:

- **Data Sources:** The specific tables and fields will vary depending on the Microsoft services you are using (e.g., Azure Storage, SharePoint, OneDrive) and whether you have enabled diagnostic logging. You might need to explore tables like `AuditLogs`, `StorageFileLogs`, `SharePointActivity`, `OfficeActivity`, `AzureNetworkAnalytics_CL`, etc.
- `<HighDownloadThreshold>, <HighResourceAccessThreshold>, <HighFileActionThresholdAfterLogin>`: Set appropriate thresholds based on your understanding of normal user activity.
- `<TopN>`: When analyzing network traffic, look at the top N entries with the highest traffic volume.
- **Correlation:** Combining login patterns with file access patterns (as shown in

the last example) can provide stronger indicators of suspicious activity.

- **Context is Crucial:** Data exfiltration attempts can often mimic legitimate activity. Understanding the user's role, typical workflows, and the sensitivity of the accessed data is essential for accurate identification.
- **Alerting and Automation:** Once you identify relevant patterns, you can create alerts in Microsoft Sentinel to be notified of such activity in the future. You can also automate responses using playbooks.

### 3. Unauthorized File Access:

This focuses on identifying access to files or folders that a user should not have permissions to.

#### Code snippet

- // Look for "Access Denied" or similar events in file access logs
- StorageFileLogs // Replace with the relevant table

- | where UserId == "<UserPrincipalName>"  
and AccessResult == "Denied" // Adjust  
AccessResult based on your logs
- | summarize DeniedAccessCount = count()  
by UserId, ObjectName
- | where DeniedAccessCount >  
<DeniedAccessThreshold> // Set a threshold for  
the number of denied access attempts to the  
same object
- | project TimeGenerated, UserId,  
ObjectName, DeniedAccessCount
- 
- // Look for successful access to  
sensitive or restricted files/folders by users  
who don't normally access them
- StorageFileLogs // Replace with the  
relevant table

- | where ObjectName startsWith "<SensitiveFilePathPrefix>" and AccessType in ("Read", "Write") // Filter for specific sensitive paths
- | summarize AccessCount = count() by UserId, ObjectName
- | where UserId !in (<ListOfAuthorizedUsers>) // Compare against a list of users who are normally authorized
- | project TimeGenerated, UserId, ObjectName, AccessType
- 
- // Look for privilege escalation followed by access to sensitive resources (requires correlating different log types)
- let elevatedPrincipals = AuditLogs

- | where OperationName contains "AssignRoleToPrincipal" and TargetResources contains "<SensitiveResource>" // Identify potential privilege escalation
- | project ElevatedUserId = InitiatedBy.user.userPrincipalName, ElevationTime = TimeGenerated;
- StorageFileLogs // Or relevant resource access logs
- | join kind=inner elevatedPrincipals on \$left.UserId == \$right.ElevatedUserId and \$left.TimeGenerated > \$right.ElevationTime and \$left.TimeGenerated < \$right.ElevationTime + 1h // Look for access shortly after elevation
- | where ObjectName startswith "<SensitiveFilePathPrefix>" and AccessType in ("Read", "Write")
- | project TimeGenerated, UserId, ObjectName, AccessType, ElevationTime

## Key Considerations:

- <DeniedAccessThreshold>: Set a threshold for the number of denied access attempts to the same resource by a user.
- <SensitiveFilePathPrefix>: Replace this with the starting path of sensitive file shares or folders.
- <ListOfAuthorizedUsers>: Maintain a list of users who are normally authorized to access specific sensitive resources.
- **Privilege Escalation**  
**Detection:** Identifying privilege escalation attempts is crucial for preventing unauthorized access. Correlate audit logs related to role assignments with subsequent access attempts.

## General Best Practices for KQL Queries:

- **Be Specific:** The more specific your criteria, the fewer false positives you will encounter.
- **Use Time Windows:** Analyze activity within relevant timeframes (e.g., last hour, last day).
- **Normalize Data:** Ensure your data sources are consistent and that user and

resource identifiers are properly normalized.

- **Parameterize Queries:** Use variables or parameters for thresholds, user lists, and file paths to make your queries more flexible.
- **Test and Tune:** Thoroughly test your queries in a non-production environment and adjust thresholds as needed to minimize noise and maximize detection accuracy.
- **Combine Indicators:** Correlating multiple suspicious behaviors can significantly increase the confidence in your findings.
- **Leverage Anomaly Detection:** Microsoft Sentinel provides built-in anomaly detection rules that can help identify unusual behavior without requiring you to write complex KQL queries from scratch. Review and enable relevant anomaly detection rules.

By combining these KQL query techniques and tailoring them to your specific environment and security concerns, you can effectively identify unusual login attempts, failed logins, and patterns indicative of potential data exfiltration or unauthorized file access. Remember that

continuous monitoring, analysis, and tuning of your queries are essential for maintaining a strong security posture.

# Chapter 64

---

## KQL Kung Fu: Real-World Challenges to Sharpen Your Skills

Welcome, KQL warriors, to the dojo of data querying! If you're ready to transform your Kusto Query Language (KQL) skills from a white belt to a black belt, this chapter is your training ground. KQL, the query language powering Azure Data Explorer, Microsoft Sentinel, and Log Analytics, is your weapon for slicing through massive datasets to uncover insights. But mastering it requires practice, creativity, and a bit of fun. Let's gamify your learning with real-world challenges, practical puzzles, and "can you solve this" scenarios that mirror the chaos of production environments. Grab your virtual nunchucks, and let's dive into the art of KQL Kung Fu!

### Why Gamify KQL Learning?

KQL isn't just about writing queries—it's about solving problems under pressure, like a cybersecurity analyst hunting for threats or a

data engineer debugging a pipeline. Gamifying your practice makes learning engaging, builds muscle memory, and prepares you for real-world scenarios. Each challenge below is a “KQL Kata”—a puzzle inspired by actual use cases, complete with sample data, a problem to solve, and hints to guide you. Solve them, and you’ll earn your KQL stripes!

## KQL Kata #1: The Suspicious Spike

**Scenario:** You’re a cybersecurity analyst. Your boss storms in: “There’s a spike in failed login attempts! Is it a brute-force attack?” You need to analyze authentication logs to find the culprit.

### Sample Data:

```
let AuthLogs = datatable(Timestamp: datetime,  
User: string, Status: string, IPAddress:  
string)
```

```
[
```

```
    datetime(2025-06-24T08:00:00), "alice",  
"Success", "192.168.1.10",
```

```
    datetime(2025-06-24T08:01:00), "bob",
"Failed", "10.0.0.5",

    datetime(2025-06-24T08:01:10), "bob",
"Failed", "10.0.0.5",

    datetime(2025-06-24T08:01:20), "alice",
"Failed", "192.168.1.10",

    datetime(2025-06-24T08:02:00), "bob",
"Failed", "10.0.0.5",

    datetime(2025-06-24T08:03:00), "charlie",
"Success", "172.16.0.1"

];
```

**Challenge:** Identify users with 3 or more failed login attempts within a 5-minute window, and list their IP addresses. Bonus points: Summarize the total failed attempts per IP.

**Can You Solve This?**

1. Filter for failed logins.
2. Group by user and time window to count attempts.
3. Find IPs associated with suspicious users.

**Hint:** Use bin() for time windows and summarize for aggregation.

**Solution** (try it first!):

```
AuthLogs
```

```
| where Status == "Failed"
```

```
| summarize FailedCount = count() by User,  
IPAddress, bin(Timestamp, 5m)
```

```
| where FailedCount >= 3
```

**Answer:** Bob has 3 failed attempts from 10.0.0.5 in the 5-minute window starting at 08:00:00.

**Level Up:** Modify the query to alert if the same IP targets multiple users. Use dcount(User) in your summarize.

## KQL Kata #2: The Performance Puzzle

**Scenario:** You're a data engineer. Your application's performance tanked last night. The ops team suspects a specific API endpoint is slowing things down. You need to find the slowest endpoints from performance logs.

### Sample Data:

```
let PerfLogs = datatable(Timestamp: datetime,  
Endpoint: string, DurationMs: int)
```

```
[
```

```
    datetime(2025-06-24T01:00:00),  
"/api/orders", 200,
```

```
        datetime(2025-06-24T01:01:00),  
        "/api/users", 150,  
  
        datetime(2025-06-24T01:01:30),  
        "/api/orders", 800,  
  
        datetime(2025-06-24T01:02:00),  
        "/api/products", 300,  
  
        datetime(2025-06-24T01:02:10),  
        "/api/orders", 750  
    ] ;
```

**Challenge:** Find the top 2 endpoints with the highest average duration, but only consider endpoints with at least 3 calls. Display the average and maximum duration.

### Can You Solve This?

1. Group by endpoint and count calls.
2. Filter for endpoints with 3+ calls.
3. Calculate average and max duration, then sort.

**Hint:** Use summarize with avg() and max(), and top to limit results.

## Solution:

```
PerfLogs
```

```
| summarize CallCount = count(), AvgDuration =  
avg(DurationMs), MaxDuration = max(DurationMs)  
by Endpoint
```

```
| where CallCount >= 3
```

```
| top 2 by AvgDuration desc
```

**Answer:** /api/orders has an average duration of 583.33ms and a max of 800ms (3 calls).

**Level Up:** Add a condition to flag endpoints where the max duration exceeds 700ms. Use extend to create a flag column.

## KQL Kata #3: The Data Detective

**Scenario:** You're a business analyst. Your company's e-commerce site had a drop in sales. You suspect a specific region is underperforming. Dig into the sales logs to confirm.

### **Sample Data:**

```
let SalesLogs = datatable(Timestamp: datetime,  
Region: string, OrderAmount: double)
```

```
[
```

```
    datetime(2025-06-24T09:00:00), "North",  
100.50,
```

```
    datetime(2025-06-24T09:01:00), "South",  
75.25,
```

```
    datetime(2025-06-24T09:02:00), "North",  
50.00,
```

```
    datetime(2025-06-24T09:03:00), "West",  
200.75,
```

```
    datetime(2025-06-24T09:04:00), "South",  
20.10
```

```
];
```

**Challenge:** Calculate the total and average order amount per region, and identify regions with an average order amount below \$60.

## Can You Solve This?

1. Group by region.
2. Compute total and average order amounts.
3. Filter for low-performing regions.

**Hint:** Use summarize with sum() and avg().

## Solution:

```
SalesLogs
```

```
| summarize TotalAmount = sum(OrderAmount),  
AvgAmount = avg(OrderAmount) by Region  
  
| where AvgAmount < 60
```

**Answer:** South has an average order amount of \$47.68, and North has \$75.25. Only South is below \$60.

**Level Up:** Add a time dimension to see if the low performance is recent. Use bin(Timestamp, 1h) and compare hourly trends.

## KQL Kata #4: The Anomaly Hunter

**Scenario:** You're monitoring IoT devices. One device is sending abnormal sensor readings. You need to find devices with readings that deviate significantly from their usual patterns.

### Sample Data:

```
let SensorLogs = datatable(Timestamp: datetime,  
DeviceId: string, Reading: double)
```

```
[
```

```
        datetime(2025-06-24T10:00:00), "D1", 25.0,  
        datetime(2025-06-24T10:01:00), "D1", 26.0,  
        datetime(2025-06-24T10:02:00), "D1", 100.0,  
        datetime(2025-06-24T10:00:00), "D2", 30.0,  
        datetime(2025-06-24T10:01:00), "D2", 31.0  
    ];
```

**Challenge:** Identify devices with readings that are more than 2 standard deviations above their average reading.

## Can You Solve This?

1. Calculate the average and standard deviation per device.
2. Compare each reading to the device's threshold ( $\text{avg} + 2 \times \text{stdev}$ ).

### 3. List anomalies.

**Hint:** Use summarize for stats and join to compare readings.

#### Solution:

```
let Stats = SensorLogs  
  
| summarize AvgReading = avg(Reading),  
StdevReading = stdev(Reading) by DeviceId;  
  
SensorLogs  
  
| join kind=inner Stats on DeviceId  
  
| where Reading > (AvgReading + 2 *  
StdevReading)  
  
| project Timestamp, DeviceId, Reading
```

**Answer:** Device D1 has an anomalous reading of 100.0 at 10:02:00.

**Level Up:** Add a rolling window to calculate stats over the last 10 minutes using range and prev().

## Tips to Master Your KQL Kung Fu

1. **Practice Regularly:** Use platforms like Azure Data Explorer's free cluster or Microsoft Sentinel's sample data to experiment.
2. **Break It Down:** Decompose complex problems into smaller steps (filter, group, join).
3. **Explore Functions:** Learn KQL's powerful functions like series\_stats, make-series, and arg\_max for advanced analytics.
4. **Join the Community:** Check out Microsoft's KQL forums or X posts (search #KQL) for tips and real-world queries.
5. **Think Like a Detective:** Always ask, "What story is the data telling me?"

## Your KQL Dojo Awaits

These KQL Katas are just the beginning. Each challenge mimics the messy, high-stakes problems you'll face in the wild—whether you're hunting threats, optimizing systems, or chasing business insights. Try tweaking the queries,

inventing your own datasets, or combining challenges (e.g., correlate failed logins with slow API calls). Share your solutions or ask for hints on X with #KQLKungFu—let's build a community of query ninjas!

Ready to level up? Pick a Kata, fire up your KQL editor, and show the data who's boss. What's your next move, sensei?



---

## Extra

If you're new to KQL, check out Microsoft's [KQL tutorial](#) for a quick start. For real-time practice, search X for #KQL challenges or post your own!

# Chapter 65

---

## The Must Learn KQL Workshop: Empowering You to Build Custom Learning Experiences

Hello, fellow data enthusiasts and security pros! If you've been following my journey with the Kusto Query Language (KQL), you know how passionate I am about making this powerful tool accessible to everyone. So, I'm also thrilled to share an exciting update to the [Must Learn KQL](#) Workshop. I've revamped the workshop page to incorporate the latest Workshop Series modules, giving you the flexibility to mix and match content and create tailored KQL workshops that fit your unique needs. Whether you're training your team, educating customers, or just diving deeper yourself, this update puts the power in your hands.

Let's dive into the details—what this means, why it matters, and how you can get started.

# The Big Update: Introducing Mix-and-Match Workshop Modules

Now, onto the star of this post—the updated [Must Learn KQL Workshop](#)! Previously, the workshop was a structured set of materials designed for guided learning or team sessions. But I've listened to your feedback: many of you wanted more customization to align with specific scenarios, like focusing on security analytics or data visualization.

That's why I've integrated the new [Workshop Series modules](#) directly into the workshop page. These modules build on the core series but are now modularized for flexibility. You can pick and choose based on your goals, audience, or time constraints. Think of it as a KQL buffet—select the appetizers, mains, and desserts that suit your taste!

## What's in the Workshop Series Modules?

Drawing from the foundational series, the modules cover key KQL concepts in bite-sized, self-contained units. Here's a glimpse of what you might find (based on the evolving series structure):

Module Category	Examples of Topics	Ideal For
Basics	Tools & Resources, Workflow, Search Operators	Beginners getting started with KQL syntax and environments.
Operators Essentials	Where, Count, Summarize, Extend, Project	Building efficient queries for filtering and aggregating data.
Advanced Techniques	Join, Union, Let Statements, Render with Visualizations	Complex data correlations and custom analytics rules.
Practical Applications	Building Sentinel Rules, Schema Exploration, Interface Tips	Real-world use in Microsoft security tools like Sentinel and Defender.

Each module includes:

- Explanatory text and examples.
- Sample queries to copy-paste and run.
- Hands-on activities using demo environments (no setup required for basics).
- Links to corresponding videos and blog posts for deeper dives.

The new additions emphasize modularity, with updated content reflecting KQL's evolution (like enhancements for AI integrations and advanced editions released in 2025).

## How to Mix and Match for Your Own Custom Workshop

Creating a custom workshop is straightforward:

1. **Head to the GitHub Repo:** Navigate to <https://github.com/rod-trent/MustLearnKQL/tree/main/Workshop>.
2. **Browse the Modules:** Review the directory for individual module files.
3. **Select and Sequence:** Choose modules that align with your objectives. For a beginner session, start with Basics and Operators. For a security-focused workshop, prioritize Practical Applications.
4. **Customize:** Edit the content if needed—add your own examples, branding, or company-specific data scenarios.
5. **Deliver:** Use the materials for in-person training, virtual sessions, or self-paced learning. Pair with the demo environment for interactive practice.

Prerequisites are minimal: Access to a KQL-compatible tool like the Azure portal or Log Analytics demo workspace. No advanced setup is required, making it ideal for quick starts.

## Why This Update Matters

In a world where data volumes explode daily, mastering KQL isn't just a skill—it's a superpower for cloud security, operations, and analytics. By making the workshop modular, we're empowering you to:

- **Personalize Learning:** Tailor content to your team's skill level or industry focus.
- **Scale Training:** Easily adapt for large groups, customers, or ongoing education programs.
- **Stay Current:** Incorporate the latest KQL features, including those from advanced series like KQL Mysteries.
- **Give Back:** Remember, resources like the book and merch support St. Jude—learning while contributing to a great cause.

Feedback from early adopters has been fantastic, and I've even added a form for sharing your experiences and attendee numbers.

Download, mix, match, and let me know how it goes—drop a tweet with #MustLearnKQL, or email feedback.

## The Workshop Series

The following represents additional material that you can use to incorporate into your own workshops for delivery to your customers and your organizations. Part of the *Advanced Must Learn KQL* series, each post highlights specific advanced operators and plugins and contains its own dataset and instructions for using them.

Pick one. Pick several. Combine them. Or, like a Pokemon master - collect 'em all!

- [\*\*Gamifying KQL: Build Your Own Query Challenge\*\*](#)
- [\*\*KQL Game Show: Query or Bust!\*\*](#)
- [\*\*KQL Band Tour: Rocking the Data Stage\*\*](#)
- [\*\*KQL Wildlife Safari: Tracking Data Beasts\*\*](#)
- [\*\*KQL Haunted Database: Ghostly Queries\*\*](#)
- [\*\*KQL Space Odyssey: Querying the Cosmos\*\*](#)
- [\*\*KQL Superhero Academy: Power Up Your Queries\*\*](#)
- [\*\*KQL Time Heist: Stealing Insights from History\*\*](#)

- **KQL Reality Show: Survivor: Data Island**
- **KQL Fantasy Quest: The Query of Legends**
- **KQL Courtroom Drama: The Data Trial**
- **KQL Pirate Adventure: Hunt for the Data Treasure**
- **KQL Choose-Your-Own-Adventure: Interactive Query Learning**
- **KQL Art Gallery: Visualizing Data with Creative Queries**
- **KQL Song Parodies: Learn Queries Through Music**
- **KQL Cooking Show: Recipes for Tasty Data Insights**
- **KQL Time Machine: Querying Historical Events to Learn Data Analysis**
- **KQL Celebration: Using Queries to Fulfill Independence Dreams**
- **KQL Escape Room: Break Out with Your Query Skills**

- **KQL vs. the Zombie Apocalypse: Surviving with Data Queries**
- **KQL in Pop Culture: Querying Your Favorite Fictional Worlds**
- **KQL Treasure Hunt: Solve a Data Mystery with Queries**
- **KQL Game Night: Data Board Game**
- **KQL Rock Anthem Memory Match**
- **KQL Fashion Show: Strutting the Data Runway**
- **KQL Sports League: Query for the Win**
- **KQL Sci-Fi Lab: Experimenting with Data**
- **KQL Time Machine: Querying History's Data**
- **KQL Quest: The Data Detective's Adventure**
- **KQL Spy Mission: Decoding the Data Conspiracy**
- **KQL Circus Extravaganza: Juggling Data Acts**

- **KQL Superhero Smackdown: Query vs. Chaos**
- **KQL Detective Agency: The Case of the Missing Data**

## Get Started Today!

Ready to build your custom KQL workshop? Jump over to the updated page on GitHub: <https://github.com/rod-trent/MustLearnKQL/tree/main/Workshop>.

## Learn more

- **Must Learn KQL** - the blog series, the book, the completion certificate, the video channel, the merch store, the workshop, and much more... <https://aka.ms/MustLearnKQL>
- **The Definitive Guide to KQL**: Using Kusto Query Language for operations, defending, and threat hunting <https://amzn.to/42JRsCL>



# Shameless Plug



Sure - I'm a seasoned technical book author, known for more recent works on Security Copilot and KQL, but many may not know I also write bestselling fiction.

Writing, whether technical or fictional, is my passion—it calms me. My fiction offers readers a way to unwind from the chaos of professional life. Studies show just six minutes of reading daily can reduce stress by 68%.

I'm also an inventor at heart. Writing stories is a form of invention, but I'm always pushing boundaries and challenging norms.

So, I'm excited to announce my latest invention: *Rod's Fiction Universe*

**<https://Rod'sFictionBooks.com>**

*Rod's Fiction Universe* is where you can dive into a world of imagination and storytelling! By subscribing here, you gain exclusive access to my entire collection of fiction works—past, present, and future.

Explore every page of my published fiction books, get first looks at new releases, and follow along with my creative journey as I share sneak peeks and full expanses of works in progress (hint: read chapters as they are written!). Plus, connect with me directly and share your thoughts and help build storylines and characters by commenting on the chapters as they take shape.

Whether you're a longtime fan or a new reader, this is your gateway to immersive stories, fresh ideas, and the magic of fiction as it unfolds. Join now and become part of the adventure!



Welcome, data adventurers, to Must Learn KQL: Advanced Edition! Whether you're a seasoned query crafter or a curious analyst ready to level up, this book is your guide to mastering the Kusto Query Language (KQL) with confidence and flair. Building on the foundations of KQL, this advanced edition dives deep into the powerful, nuanced features of KQL, designed to transform you into a data wizard capable of taming even the wildest datasets in Azure Data Explorer and beyond.

In these pages, you'll embark on a journey through complex queries, time-series analysis, and performance optimization, all while uncovering KQL's hidden gems. Inspired by the creative spirit of past explorations—like Captain KQL's superhero saga or Archmage Kusto's epic battles against data dragons—this book blends practical expertise with engaging analogies to make advanced concepts accessible and memorable. Expect hands-on examples, real-world scenarios, and challenges that push your skills to new heights, all crafted to spark your curiosity and creativity.

The success of the original Must Learn KQL series has led to the birth of this advanced book. This book represents the culmination of everything I've worked on with KQL since the Must Learn KQL series was first released, designed to help you master KQL with real-world applications.

