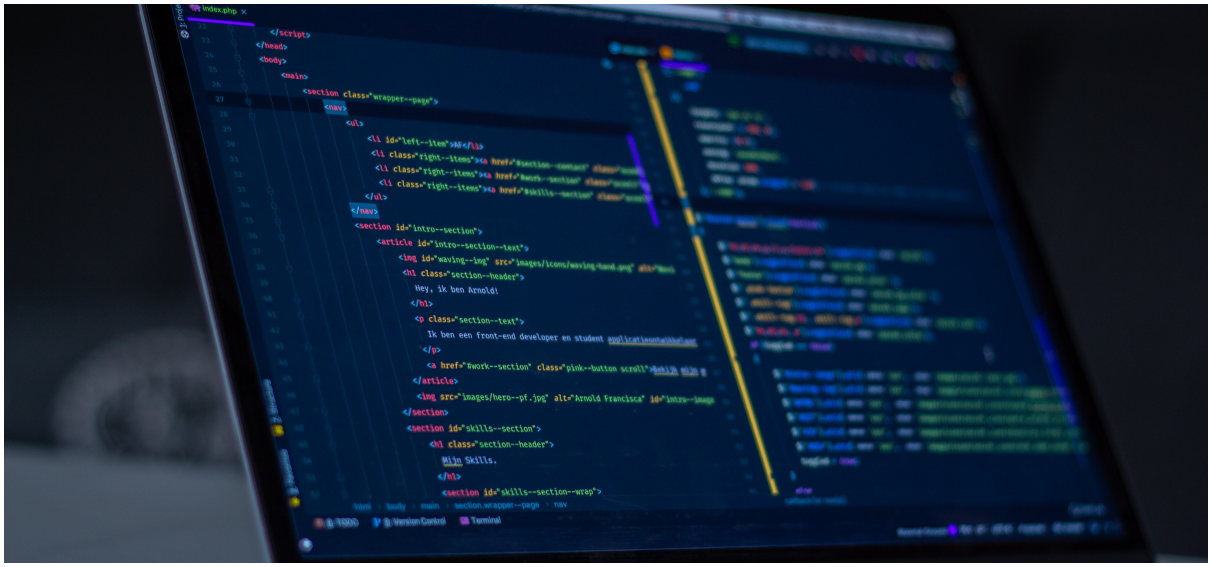


KQL Reference Manual

> -Yiannis- @sec_groundzero

Table of Contents

Disclaimer

This guide does not provide full coverage of all KQL functions and operators. The complete KQL documentation by Microsoft can be found here. The queries presented below are my own interpretation of the KQL capabilities and i take no responsibility for any errata in my understanding. Any KQL hunting queries provided below are for illustrative purposes only. Much of the operators narrative has been paraphrased from Microsoft.

Credits

I relied a lot on the work of others for this manual and the following people/sources need to be credited.

@DebugPrivilege

@rpargman

@olafhartong

@falconforceteam

@Binary_Defense

@randyfsmith

Introduction

KQL Queries Best Practices

- Use time filters first
- Use the *has* operator instead of the *contains* operator as the later searches in substrings as well
- Use case sensitive operators *==* vs *=~* when possible
- For new queries use a small *limit* at the beginning
- Use *Col =~ "lowercasestring"* instead of *tolower(Col) == "lowercasestring"*
- Select the table with the fewer rows to be the first one (left-most in query).

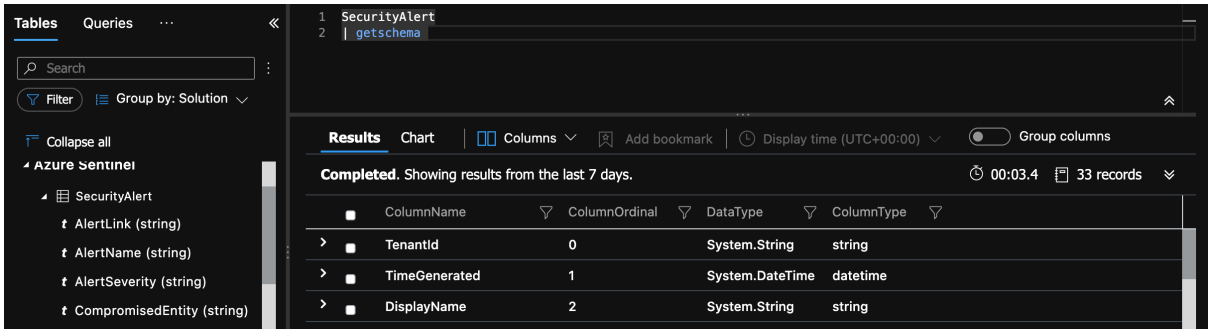
KQL Databases and Tables

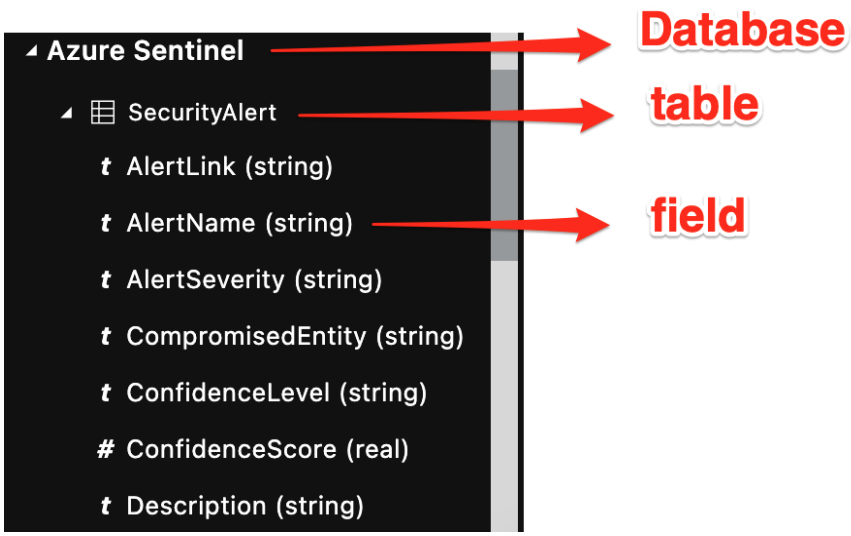
> KQL databases are divided into tables, columns, stored functions and external tables.

The tables can be either viewed from the left hand side or query their schema directly

```
SecurityAlert
| getschema
```

The *getschema* operator returns the columns and their types included in the database requested.





If looking to see the columns for a specific event

```
Sysmon
| where EventID == 1
| getschema
```

The query above will return only the columns associated with the specific event ID.

Tabular Operators

Tabular operators are operators that are executed against "tabular data" (data in one or more rows/columns)

Searching

It is possible to search for specific strings across a database or across a specific table. This will search in the Sysmon database across all columns for a case insensitive match.

```
Sysmon
| search "regsvr32"
```

The search can be customized to be case sensitive

```
Sysmon
| search kind=case_sensitive "Dns"
```

As searching can produce many results, it can be reduced with the *limit* or *take* operators.

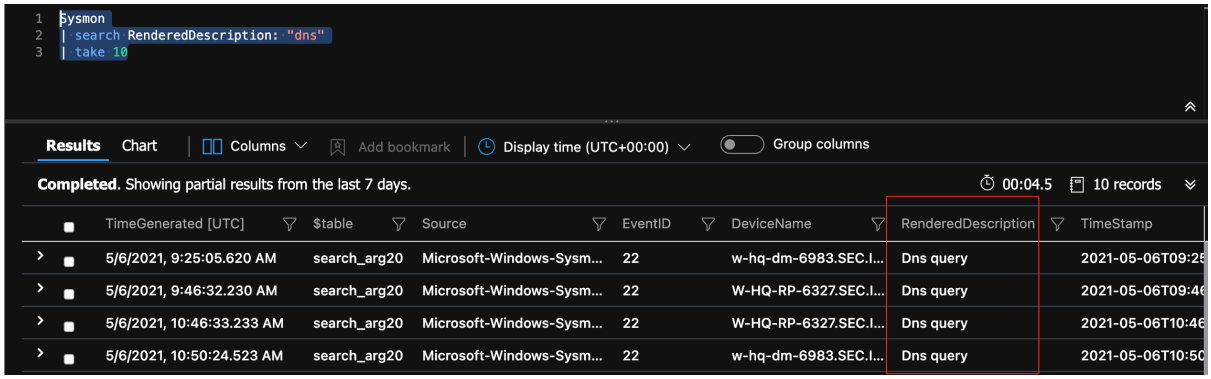
```
Sysmon
| search kind=case_sensitive "Dns"
| limit 10
```

It is also possible to search in multiple tables concurrently for a string or a value.

```
search in (Sysmon, Event, Alert) "dns"
```

Or search in a specific column directly

```
Sysmon
| search RenderedDescription: "dns"
| take 10
```



Wildcards can also be used in searches.

```
Sysmon
| search * startswith "Dns"
```

In the example above the query will look in all columns for a string that starts with "Dns".

Where

The *where* operator needs as an input a tabular input such as Sysmon or Security logs and performs filtering based on the parameters provided.

```
Sysmon
| where EventID == 1
```

The above is read as: query the Sysmon data and return the results if the EventID matches "1".

The *where* operator can also be combined to include additional queries.

```
Sysmon
| where EventID == 1 and DeviceName == "Demo"
```

The *where* operator is very useful to first limit the query to the timeperiod of interest before running other queries. This will reduce the results first by the timestamp and then perform any additional actions effectively reducing the processing time.

```
SecurityEvent
| where TimeGenerated >= ago(48h)
  and EventID == 4624
```

Different operators can be combined as well. The query is used to query the Security Event table where the *TimeGenerated* is 48 hours ago and the EventID could be either 4624 or 4688.

```
SecurityEvent
| where TimeGenerated >= ago(48h) and (EventID == 4624 or EventID == 4688)
```

If the parenthesis was not used the query would be executed as combination between the TimeGenerated and EventID 4624 or where the EventID is equal to 4688 ignoring the TimeGenerated. This is because the AND operator is between the TimeGenerated and EventID 4624.

```
SecurityEvent
| where TimeGenerated >= ago(48h) and EventID == 4624 or EventID  == 4688
| project EventID
```

The *ago* operator is extremely useful as it is used to define the period of interest. The ago operator can be defined to use the below:

- d -days
- h - hours
- m - minutes
- s - seconds
- ms - milliseconds

```
SecurityEvent
| where TimeGenerated >= ago(2d)
```

The above is read as: return all entries from the SecurityEvents table where the TimeGenerated is equal or larger than 2 days ago. Effectively this will return all entries from 2 days ago until the day the query is executed.

> 💡 *TimeGenerated* is a field contained in the SecurityEvent table. Different log sources may use a different name for the time of an event but this information can be found with the *getschema* operator.

A specific time range between dates can be used using the *between* operator.

```
Sysmon
| where TimeGenerated between (datetime(2020-10-01) .. datetime(2021-04-01))
| where EventID == 1
```

The *datetime* data type will convert the input date into a date time format as follows "10/1/2020, 12:00:00.000 AM"

Or the *now()* operator can be used to define the current timestamp as the end date.

```
Sysmon
| where TimeGenerated between (datetime(2020-10-01) .. now())
| where EventID == 1
```

To take it a step further we can adjust exactly how long before the current time the query should look for.

```
Sysmon
| where TimeGenerated between (datetime(2020-10-01) .. now(-2h))
| where EventID == 1
```

The previous query will return all entries from the Sysmon table where the time of when the entry was created is between 2020-10-01 and 2 hours before the time when the query was executed. It is important to note that the results will change as time passes by therefore using specific dates for the between *operator* may be more accurate for queries that are going to be executed again in the future.

The same method is followed when certain date ranges must be excluded.

```
Sysmon
| where TimeGenerated !between (datetime(2020-10-01) .. now(-2h))
| where EventID == 1
```

The *startofday* and *endofday* operators can be used to ensure that the entire days are included when searching using datetimes.

```
SecurityEvent
| where TimeGenerated between (startofday(datetime(2020-10-01)) .. endofday(datetime(2
021-04-01)))
```

Using the same rationale any entries during that period can be excluded by using the *!* (not) operator which in this case it will reverse the results and present all rows outside the specified period.

```
SecurityEvent
| where TimeGenerated !between (startofday(datetime(2020-10-01)) .. endofday(datetime
(2021-04-01)))
```

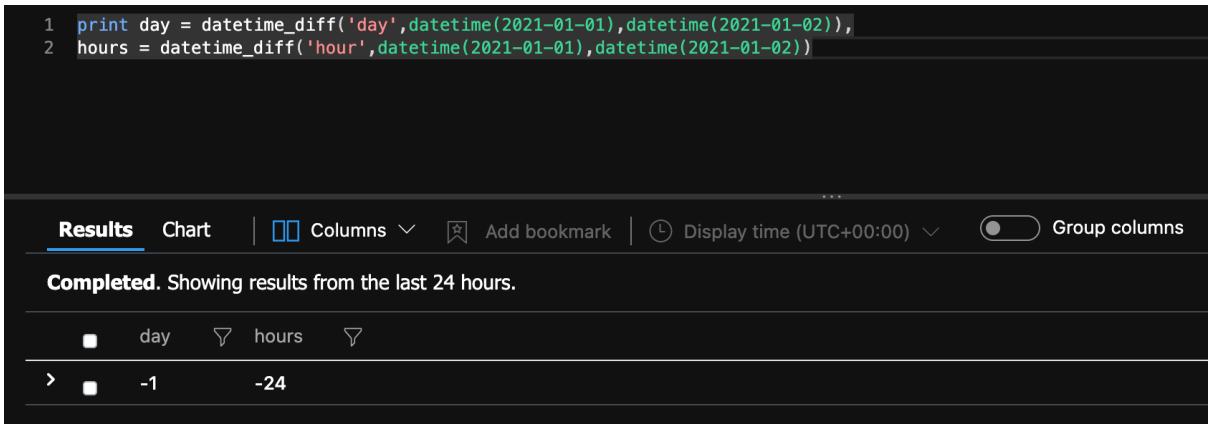
The *datetime_diff* calculates the different between two datetimes.

The periods that can be used are

- Year
- Quarter
- Month
- Week
- Day
- Hour
- Minute
- Second
- Millisecond
- Microsecond

- Nanosecond

```
print day = datetime_diff('day',datetime(2021-01-01),datetime(2021-01-02)),
hours = datetime_diff('hour',datetime(2021-01-01),datetime(2021-01-02))
```



The query outputs the duration in days and hours between the start date and the end date. This essentially subtracts the end date from the start date to calculate the output.

The *abs* operator can be used to eliminate the negative sign. Alternatively the start date can be subtracted from the end date instead.

```
print day = abs(datetime_diff('day',datetime(2021-01-01),datetime(2021-01-02))),
hours = abs(datetime_diff('hour',datetime(2021-01-01),datetime(2021-01-02)))
```

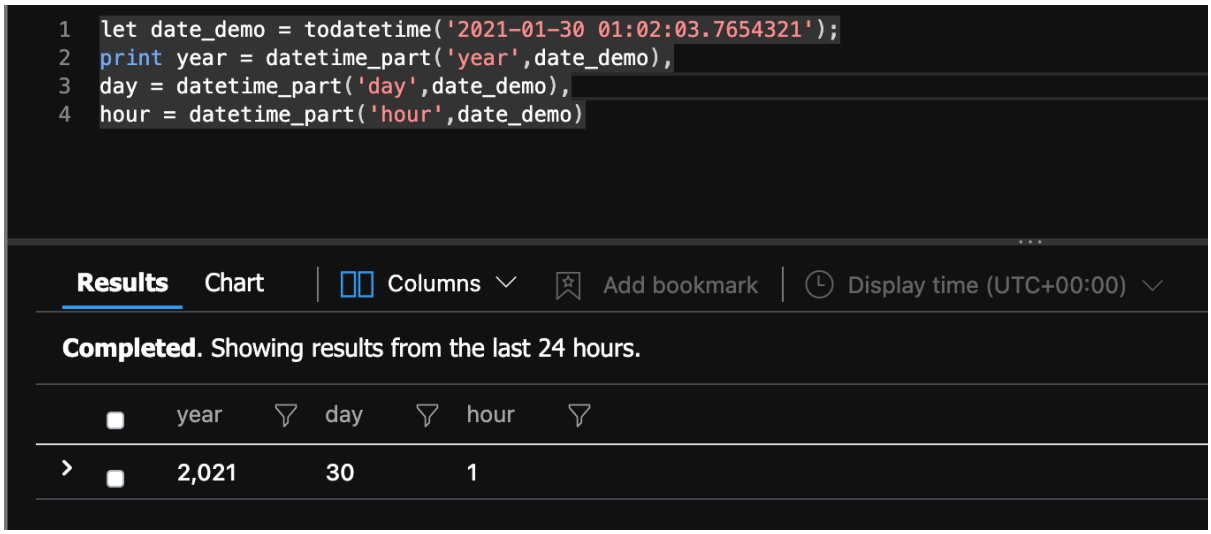
It is possible to extract certain elements from a datetime. The elements that can be extracted are:

- Year
- Quarter
- Month

- week_of_year
- Day
- DayOfYear
- Hour
- Minute
- Second
- Millisecond
- Microsecond
- Nanosecond

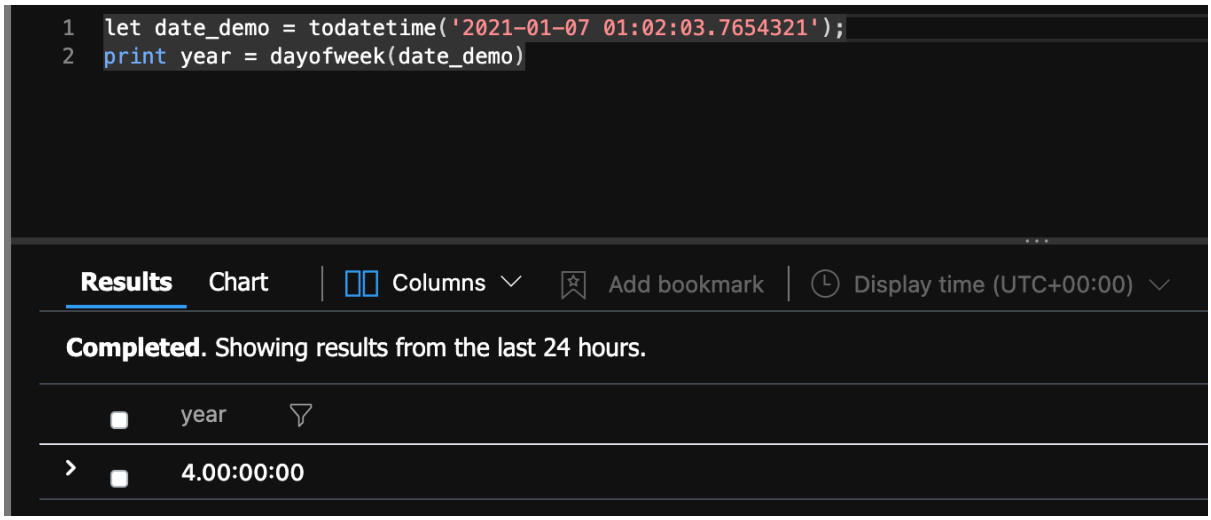
> ⚠ The let statement can be used to represent a constant value which can be used later on in the query.

```
let date_demo = todatetime('2021-01-30 01:02:03.7654321');
print year = datetime_part('year',date_demo),
day = datetime_part('day',date_demo),
hour = datetime_part('hour',date_demo)
```



It is also possible to find out the day of the week using the *dayofweek* as an offset from Sunday.

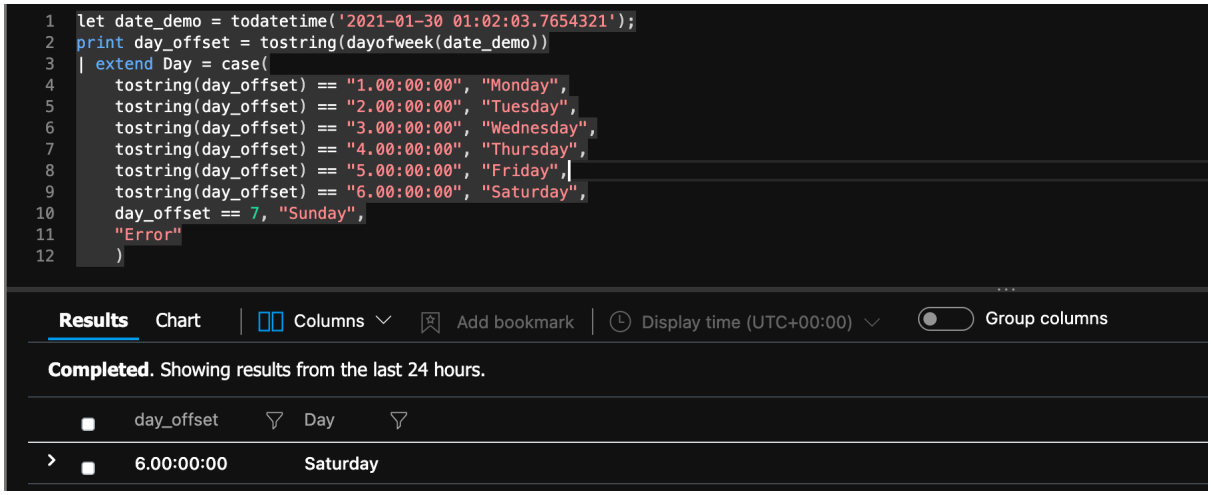
The example below returns 4 meaning 4 days after Sunday i.e Thursday



Case

The example above can be used to showcase the *case* operator. As the day returned in the previous query is in timespan format it is first converted to string and evaluated using the *case* operator. The *Error* string in the bottom of the query is the default value that will be returned in case none of the values is matched.

```
let date_demo = todatetime('2021-01-30 01:02:03.7654321');
print day_offset = tostring(dayofweek(date_demo))
| extend Day = case(
    tostring(day_offset) == "1.00:00:00", "Monday",
    tostring(day_offset) == "2.00:00:00", "Tuesday",
    tostring(day_offset) == "3.00:00:00", "Wednesday",
    tostring(day_offset) == "4.00:00:00", "Thursday",
    tostring(day_offset) == "5.00:00:00", "Friday",
    tostring(day_offset) == "6.00:00:00", "Saturday",
    day_offset == 7, "Sunday",
    "Error"
    )
```



Although most of the operators below are not described yet it is easy to see the value of the datetime operations.

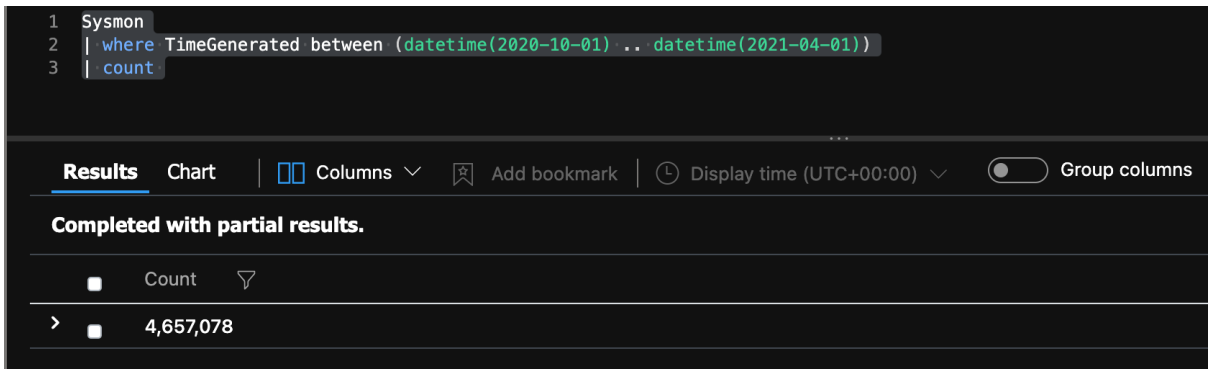
```
let proc_create = Sysmon
| where TimeGenerated >= ago(48h)
| where EventID == 1
| extend CreateTime = TimeGenerated
| project CreateTime, ProcessGuid, FileName;
let proc_end = Sysmon
| where EventID == 3
| extend EndTime = TimeGenerated
| project EndTime, ProcessGuid;
proc_create
| join (proc_end) on ProcessGuid
| extend ProcessDuration  = (EndTime-CreateTime)
| project ProcessDuration, FileName
| sort by ProcessDuration desc
```

The query above searches in two different EventIDs and calculates the duration of a process from when it was started (EventID 1) and when the process was ended (EventID 3).

Count

The *count* operator can be used to count the number of rows returned by a query.

```
Sysmon
| where TimeGenerated between (datetime(2020-10-01) .. datetime(2021-04-01))
| count
```




```
Sysmon
| where TimeGenerated between (datetime(2020-10-01) .. datetime(2021-04-01)) and FileN
ame contains "regsvr32"
| count
```

Other operators can also be combined in order to reduce the output.

Top

The *top* returns the N results specified by the query

```
Sysmon
| where TimeGenerated  >= ago(2d)
| top 5 by AccountName desc nulls last
```

This query will return the top 5 results based on the *AccountName* column which is sorted in descending order. In case the AccountName column contains any null entries these are going to be at the end of the output indicated by the *null last.*

Project

The *project* operator allows the user to control which columns to present, rename or remove in the output

```
Sysmon
| where TimeGenerated  >= ago(2h)
| where EventID ==1
| sample 5
| project AccountName
```

```
Sysmon
| where TimeGenerated  >= ago(2h)
| where EventID ==1
| sample 5
| project AccountName, computer=DeviceName
```

It is also possible to use calculated columns with the *project* operator.

```
Sysmon
| where TimeGenerated  >= ago(2h)
| where EventID ==1
| sample 5
| project AccountName, computer=DeviceName, simpleCalc=5*5
```

The example above prints the *AccountName* column and also renames the *DeviceName* column to *computer* and creates a new calculated column containing the product of the multiplication. A column can be removed from the output using the *project-away* operator.

A column can also be renamed before displayed using the *project-rename* operator.

```
Sysmon
| where TimeGenerated  >= ago(2h)
| where EventID ==1
| project-rename Computer = DeviceName
```

Extend

The *extend* operator creates a calculated column and append them to the result set.

```
Sysmon
| where TimeGenerated  >= ago(2h)
| where EventID ==1
| extend Computer = DeviceName, hours_ago = now() - TimeGenerated
```

In the example above, a new column will be created named *hours_ago* will include the time difference between the time of the execution of the query and the time when the event was generated.

Summarize

The *summarize* operator is extremely useful as it can help give meaning to the results when presenting aggregated data. There are many different types of aggregations including but not limited to:

- count
- any
- avg
- dcount
- dcountif
- max
- min
- stdev
- sum
- variance
- ...

```
Sysmon
| where TimeGenerated between (datetime(2020-10-01) .. datetime(2021-04-01)) and FileN
ame has "regsvr32"
| summarize count() by AccountName
```

The previous query will create a summary of the *AccountName* column by counting the values for each entry in the column.

Counts can also be performed on multiple columns at the same time. In the query below the *count* operator will return the counts for the *AccountName* and *EventID* columns.

```
Sysmon
| where TimeGenerated between (datetime(2020-10-01) .. now())
| summarize count() by AccountName,EventID
```

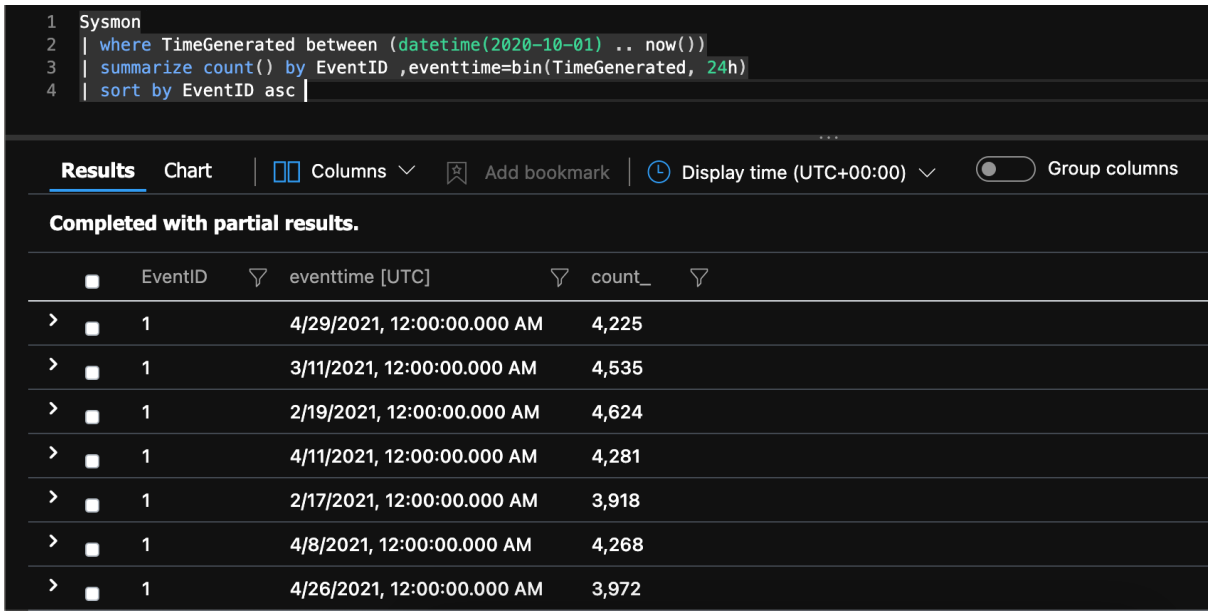
The example below uses the *min* and *max* operators to find the lowest(earliest) and highest (latest) datetime of events where the *FileName* column contained the "*regsvr*" string.

```
Sysmon
| where TimeGenerated between (datetime(2020-10-01) .. datetime(2021-04-01)) and FileN
ame has "regsvr32"
| summarize Min = min(TimeGenerated), Max = max(TimeGenerated)
```

The *bin* operator is very helpful to create buckets or groups between a scattered set of values as they will be grouped into a smaller set of specific values.

```
Sysmon
| where TimeGenerated between (datetime(2020-10-01) .. now())
| summarize count() by EventID ,eventtime=bin(TimeGenerated, 24h)
```

The query above will perform a count by *EventID* and then create 24-hour (daily) long bins based on the *TimeGenerated* field.



This example presents the distribution of EventIDs on a daily basis.

Datatable

The *datatable* operator returns a table whose schema and values are defined in the query itself. In datatables the column name and type must be defined as well as the value type.

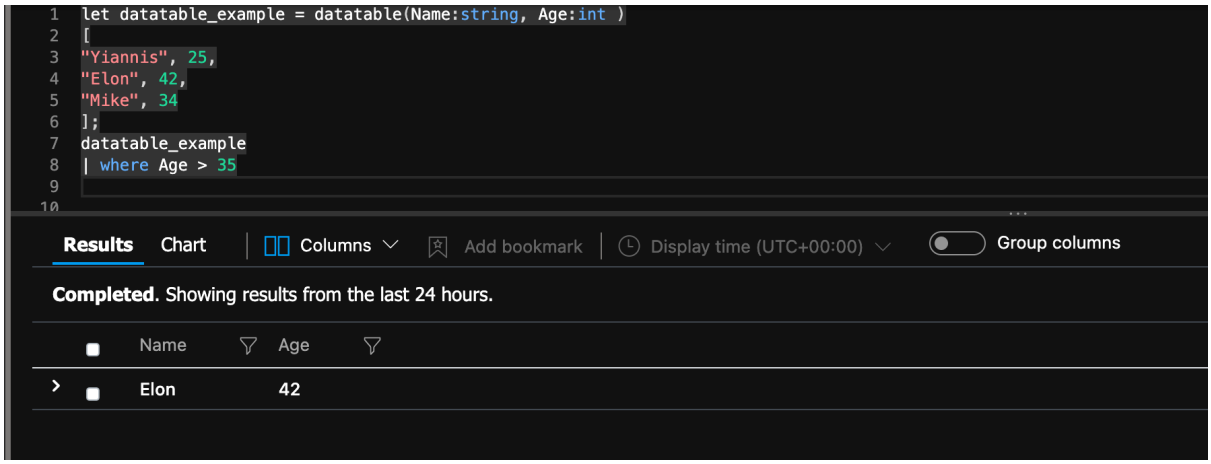
In the following datatable two columns are created. The first one holds the name of the person and it is of type string and the second holds the age of the

person and it is type integer.

```
let datatable_example = datatable(Name:string, Age:int )
[
"Yiannis", 25,
"Elon", 42,
"Mike", 34
]
```

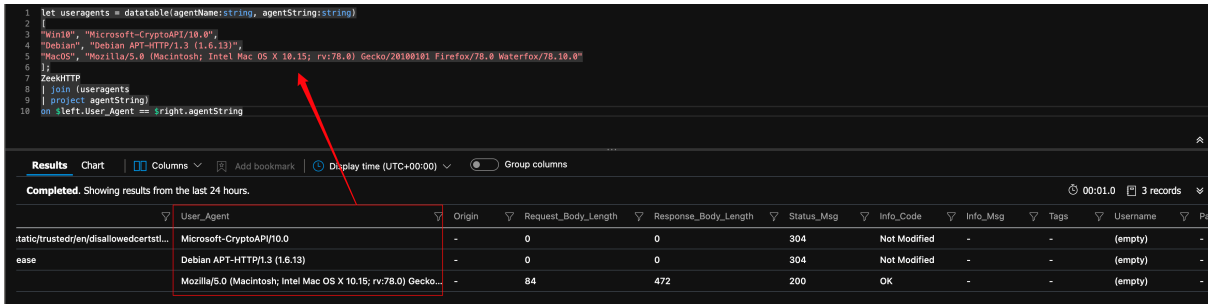
The datatable can then be incorporated in the query to search for example inside the datatable for values in the age column which are higher than 35.

```
let datatable_example = datatable(Name:string, Age:int )
[
"Yiannis", 25,
"Elon", 42,
"Mike", 34
];
datatable_example
| where Age > 35
```



A more useful example would be to search the list of known user agents against the HTTP events logged by Zeek.

```
let useragents = datatable(agentName:string, agentString:string)
[
"Win10", "Microsoft-CryptoAPI/10.0",
"Debian", "Debian APT-HTTP/1.3 (1.6.13)",
"MacOS", "Mozilla/5.0 (Macintosh; Intel Mac OS X 10.15; rv:78.0) Gecko/20100101 Firefo
x/78.0 Waterfox/78.10.0"
];
ZeekHTTP
| join (useragents
| project agentString)
on $left.User_Agent == $right.agentString
```



Dynamic

The *dynamic* operator is a scalar data type that can have as values:

- bool, datetime, guid, int, long, real, string, and timespan
- Array of dynamic values

```
let timeframe = 2d;
let lolbins = dynamic(["advpack.dll","at.exe","atbroker.exe","bash.exe","bitsadmin.ex
e","Comsvcs.dll","devtoolslauncher.exe","diantz.exe","diskshadow.exe","dllhost.exe","D
nscmd.exe","dxcap.exe","Esentutl.exe","eventvwr.exe","expand.exe","explorer.exe","extr
ac32.exe","findstr.exe","forfiles.exe","ftp.exe","Gpscript.exe","hh.exe","Ie4uinit.ex
e","Ieadvpack.dll","ieframe.dll","Infdefaultinstall.exe","makecab.exe","manage-bde.ex
e","Presentationhost.exe","presentationhost.exe","print.exe","psr.exe","Rasautou.ex
e","reg.exe","regedit.exe","regini.exe","Register-cimprovider.exe","regsvcs.exe","regs
vr32.exe","replace.exe","rpcping.exe","wsreset.exe","wuauclt.exe","xwizard.exe","Zipfl
dr.dll"]);
Sysmon
```

```
| where TimeGenerated >= ago(timeframe)
| where EventID ==1
| extend lolpath = parse_path(ProcessPath)
| where tolower(FileName) in (lolbins) and (tolower(lolpath.DirectoryPath) != tolower
(@"C:\Windows\syswow64") and tolower(lolpath.DirectoryPath) !contains tolower(@"C:\Win
dows\System32") and tolower(lolpath.DirectoryPath) !=  tolower(@"C:\Windows"))
//Suspicious Path refers to the path where the renamed lolbin executed from
| extend OriginalBinaryName=FileName, RenamedBinaryName = lolpath.Filename ,ExecutionL
ocation = lolpath.DirectoryPath, ProcessCommandLine
| project DeviceName, AccountName,RenamedBinaryName,OriginalBinaryName, ExecutionLocat
ion, ProcessCommandLine, TimeGenerated
| sort by DeviceName asc, TimeGenerated desc
```

Sort/Order

The *sort* operator sorts the rows of the input table into order by one or more columns. The data can be sorted either in ascending or descending order with *asc* or *desc.* In case the output contains null values these can be sorted to be at the top or bottom using the *nulls first* or *nulls last* respectively.

```
Sysmon
| where EventID == "22"
| sort by DnsQueryResults asc
```

```
Sysmon
| where EventID == "22"
| sort by DnsQueryResults desc nulls first
```

The *asc, desc* and *nulls* can be omitted. The default is *desc*. Similarly the *order* operator can be used.

Distinct

The *distinct* operator produces a table with the distinct combination of the provided columns from the input table.

```
SecurityEvent
| where EventID == 4624
| distinct Account
```

Multiple columns can also be used as input and the output will show the unique combinations between the two columns.

```
SecurityEvent
| where EventID == 4624
| distinct Account, Computer
```

```
ZeekHTTP
| where dst_port !in ("443", "80")
| distinct dst_port, Host, src_port, Uri
```

Top / Top-Hitters

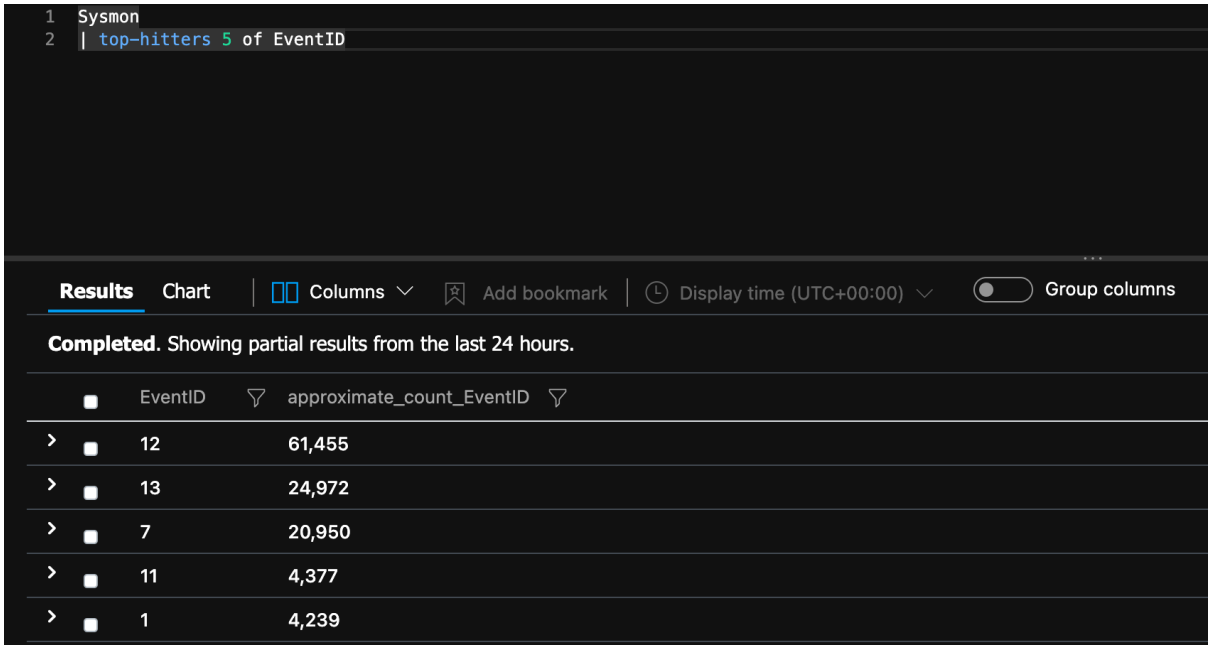
The *top* operator returns the first number of rows sorted by the specified column.

```
Sysmon
| where EventID == 1
| top 5 by DeviceName
```

The *asc* and *desc* are not mandatory by can control where the selection is made from the top or the bottom.

The *top-hitters* operator returns an approximation for the most popular distinct values, or the values with the largest sum, in the input.

```
Sysmon
| top-hitters 5 of EventID
```



This query returns the top 5 distinct values based on the EventID column. Further drill down can be done into a specific EventId and find the top-hitters there.

```
Sysmon
| where EventID == 1
| top-hitters 5 of DeviceName
```

Sample / Sample-Distinct

The *sample* operator simply returns random rows from the input table.

```
Sysmon
| where EventID == 1
| sample 3
```

The sample-distinct operator returns a single column with the specified number of distinct values.

> ⚠ This operator requires a column to be used as input

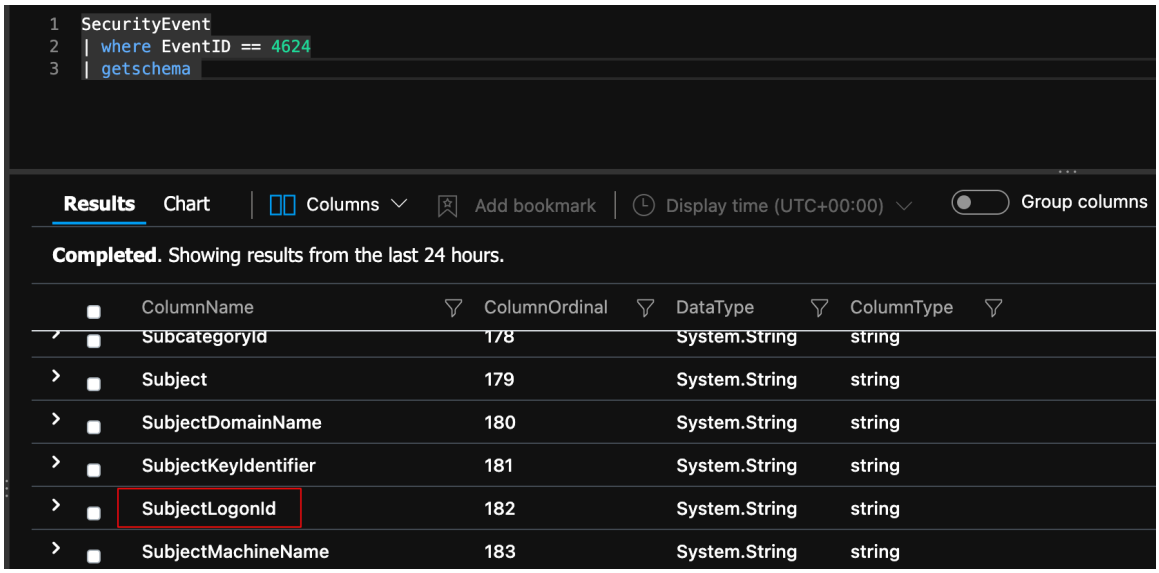
```
Sysmon
| where EventID == 1
| sample-distinct 5 of DeviceName
```

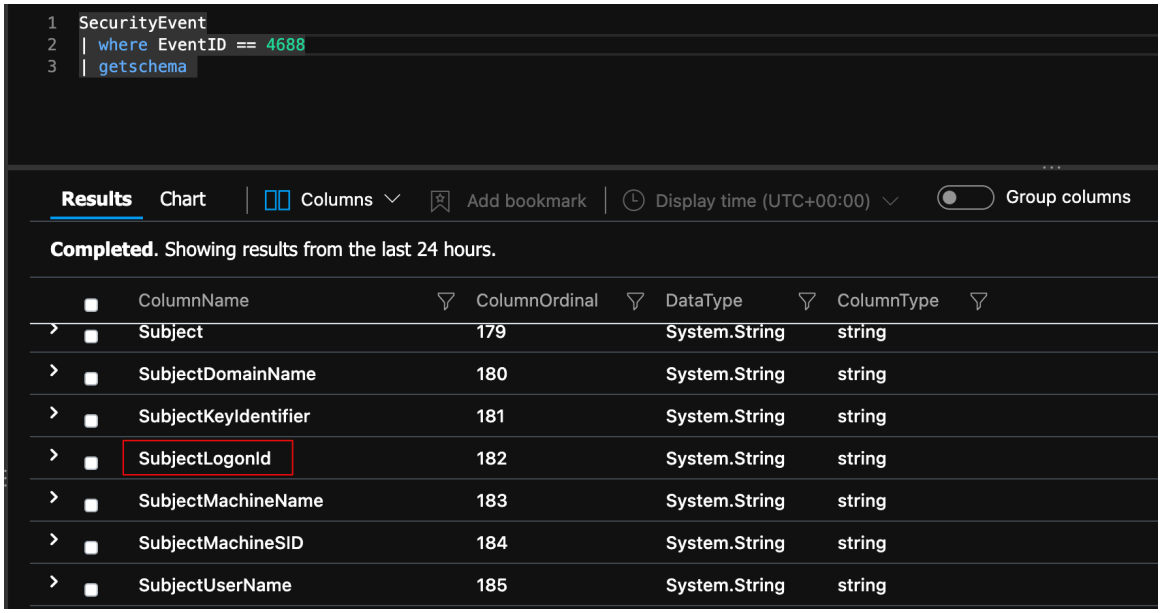
Join

The join operator is very important as it allows for joining tables together through matching values from specified columns.

Example - Joining the 4624 and 4688 EventIDs.

We first query for the schema of *4624 - An account was successfully logged on* and *4688- A new process has been created* to identify the columns we want to join on. These columns needs to have matching values in order to be joined.



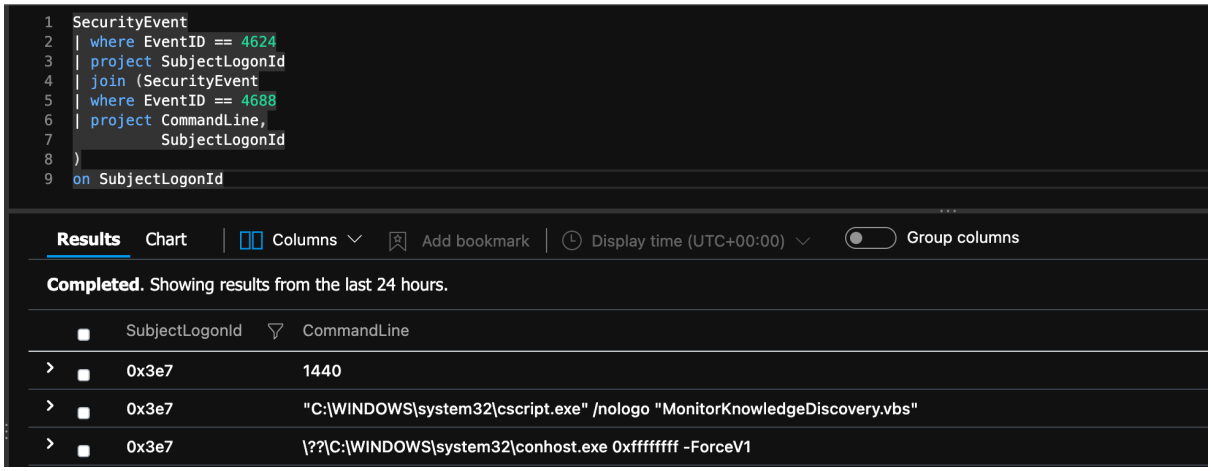


Both EventIDs have *SubjectLogonId* as a common column so that can be used.

```
SecurityEvent
| where EventID == 4624
| project SubjectLogonId
| join (SecurityEvent
| where EventID == 4688
| project CommandLine,
          SubjectLogonId
)
on SubjectLogonId
```

We begin by querying for EventID 4624 and projecting (extracting) the *SubjectLogonId*. It is important to extract all columns that should be included in the final output and especially the columns that the join will be performed on.

The tables are then joined by first querying the EventID 4688 and again projecting the useful columns for this example. Lastly the join is performed using the *on* statement and the column we want to use as the joining column.



Example - Identify how long a user was active for before they logged out.

For this example we will need both Event ID's *4624 - An account was successfully logged on* and *4647 - User initiated logoff*

```
SecurityEvent
| where TimeGenerated between (startofday(datetime(2021-03-01)) .. endofday(datetime(2
021-04-10)))
| where EventID == 4624
| where LogonType != 11
| where AccountType == "User"
| extend loginTime = TimeGenerated
| sort by loginTime desc
| project loginTime, LogonTypeName,Account,TargetLogonId, Computer
| join ( SecurityEvent
| where EventID == 4647
| extend logoutTime = TimeGenerated
| sort by logoutTime desc
| project logoutTime,TargetLogonId )
on TargetLogonId
| project loginTime,logoutTime,Account, Computer, logonTimeMinutes = datetime_diff('mi
nute',logoutTime,loginTime )
```

```
SecurityEvent
| where TimeGenerated between (startofday(datetime(2021-03-01)) .. endofday(datetime(2021-04-10)))
| where EventID == 4624
| where LogonType != 11
| where AccountType == "User"
| extend loginTime = TimeGenerated
| sort by loginTime desc
| project loginTime, LogonTypeName,Account,TargetLogonId, Computer
| join ( SecurityEvent
| where EventID == 4647
| extend logoutTime = TimeGenerated
| sort by logoutTime desc
| project logoutTime,TargetLogonId )
on TargetLogonId
| project loginTime,logoutTime,Account, Computer, logonTimeMinutes = datetime_diff('minute',logoutTime,loginTime )
```

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

Completed

loginTime [UTC]	logoutTime [UTC]	Account	Computer	logonTimeMinutes
3/29/2021, 6:43:16.243 PM	3/29/2021, 7:40:35.757 PM	[redacted]	[redacted]	57
4/8/2021, 2:44:38.120 PM	4/8/2021, 3:21:34.033 PM	[redacted]	[redacted]	37
3/12/2021, 6:21:39.337 PM	3/16/2021, 9:43:07.373 PM	[redacted]	[redacted]	5,962

Similarly as the previous example we extract the fields we are interested in and join the tables based on the *TargetLogonId*. Finally the different between the login and log out time is calculated to show the duration in minutes where the user was logged in.

For the next example the *"4688 - A new process has been created"* and *"4689 - A process has exited"* Event ID's will be used. Windows Security Event Logs don't use the same field name for the *Process ID* of the originating process and the *Process ID* of the terminated process therefore more work needs to be done. In 4688 it is called *NewProcessID* and in 4689 it is called *ProcessID*. These two values will be used as the joining string between the two events as they are unique for that process.

```
A new process has been created.

Creator Subject:
 Security ID:  SYSTEM
 Account Name:  RFSH$
 Account Domain:  LAB
 Logon ID:  0x3E7

Target Subject:
 Security ID:  LAB\rsmith
 Account Name:  rsmith
 Account Domain:  LAB
 Logon ID:  0x2C9D82

Process Information:
 New Process ID:  0x2e0e4
 New Process Name: C:\Windows\System32\RuntimeBroker.exe
 Token Elevation Type: %%1938
 Mandatory Label:  Mandatory Label\Medium Mandatory Level
 Creator Process ID: 0x268
 Creator Process Name: C:\Windows\System32\svchost.exe
 Process Command Line:
```

```
A process has exited.

Subject:

        Security ID:  WIN-R9H529RIO4Y\Administrator
        Account Name:  Administrator
        Account Domain:  WIN-R9H529RIO4Y
        Logon ID:  0x1fd23

Process Information:

        Process ID: 0xed0
        Process Name: C:\Windows\System32\notepad.exe
        Exit Status: 0x0
```

```
SecurityEvent
| where TimeGenerated between (startofday(datetime(2021-03-01)) .. endofday(datetime(2
021-04-10)))
| where EentID == 4688
| extend procID = NewProcessId
| project TimeGenerated, procID,Account,Process, SubjectUserName
| join ( SecurityEvent
| where EventID == 4689
| extend procID =  ProcessId
| project procID)
on procID
| extend ProcessStartID = procID, ProcessEndID = procID1, Username = SubjectUserName
| project TimeGenerated, ProcessStartID, ProcessEndID, Username, Process
| summarize  count() by Process, Username, eventtime=bin(TimeGenerated,1h)
```

The first step is to step the correct timeframe we want to focus on which is achieved with the *between* operator. Next we focus on the EventID we need first which is 4688 showing the details of the process that was created. In the next step we "rename" the NewProcessId field as it does not exist in the next EventID we need. At the final step we query the other EventID 4689 and again rename the *ProcessId* field to match the same we used for the other event.

Finally we join the events together based on the same process id and create summarise based on 24h long bins.

The process of renaming the columns was not necessary as in cases where the column names are not identical the tables can still be joined.

Keep in mind that the outer table is called *$left* and the inner table is called *$right* and that the join can have different flavors such as:

- *kind=leftanti*

    - Returns only the records from the left side that do not have a match on the right side
- *kind=rightanti*
    - Returns only the records from the right side that do not have a match on the left side
- *kind=leftsemi*
    - Returns all the records from the left side that have matches from the right.
- *kind=rightsemi*
    - Returns all the records from the right side that have matches from the left.
- *kind=inner*
    - Contains a row in the output for every combination of matching rows from left and right.
- *kind=leftouter*
    - Contains a row for every row on the left and right, even if it has no match. The unmatched output cells contain nulls.
- *kind=rightouter*
    - Contains a row for every row on the left and right, even if it has no match. The unmatched output cells contain nulls.

For the sake of example we will join the *4624 - An account was successfully logged on* with *Sysmon EventId 3 - Network Connect.*

The column which is going to be used for joining is called *Account* in EventID 4624 and *AccountName* is Sysmon EventId 3. In this example the outer table is going to be the EventID 4624 therefore its going to be the *$left* and Sysmon EventId 3 will be the inner table therefore *$right*.

```
SecurityEvent
| where EventID == 4624
| project Account, Computer, LogonType
| join (Sysmon
```

```
| where EventID == 3
| project AccountName, ProcessPath, ProcessId)
 on $left.Account == $right.AccountName
```

What we are testing with the query above is for users that signed in and created a network connection. Since most probably not all users have created network connections the query can be enhanced.

This query will show all users that have logged in but did not initiate any network connections indicated by the *kind=leftanti* which is going to return the results from the left side that do not have a match on the right side (inner table)

```
SecurityEvent
| where EventID == 4624
| project Account, Computer, LogonType
| join kind = leftanti (Sysmon
| where EventID == 3
| project AccountName, ProcessPath, ProcessId)
 on $left.Account == $right.AccountName
```

To do the exact opposite which is to return results only where there was a match from the left the *kind=leftsemi* is used.

```
SecurityEvent
| where EventID == 4624
| project Account, Computer, LogonType
| join kind = leftsemi (Sysmon
| where EventID == 3
| project AccountName, ProcessPath, ProcessId)
 on $left.Account == $right.AccountName
```

Union

The *union* operator merges two tables together. With *union* data is returned in different rows. If the column is common in both results then the data from both tables are put in the same column. For columns that are missing from one table the row will be empty. If we want to see the source for each field the *withsource* argument can be used to create a new column named *SourceTable* and put the name of the table in there.

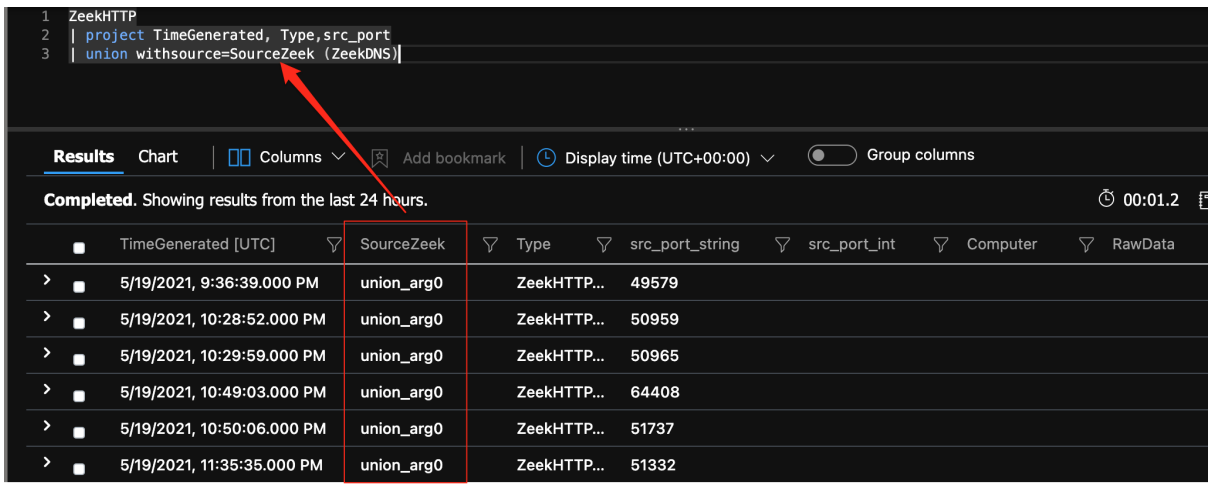
Just like the *join* operator the *kind* argument can be used to define the output. The kind can be either:

- *Inner*
  - Returns only columns which are common in both tables
- *Outer*
  - The default kind which includes all the columns from both tables

```
ZeekHTTP
| union (ZeekDNS)
```

The *withsource* argument will display which table contributed to the output. The *arg0* refers to the outer table and *arg1* to the inner table.

```
ZeekHTTP
| union withsource=SourceZeek (ZeekDNS)
|take 10
```



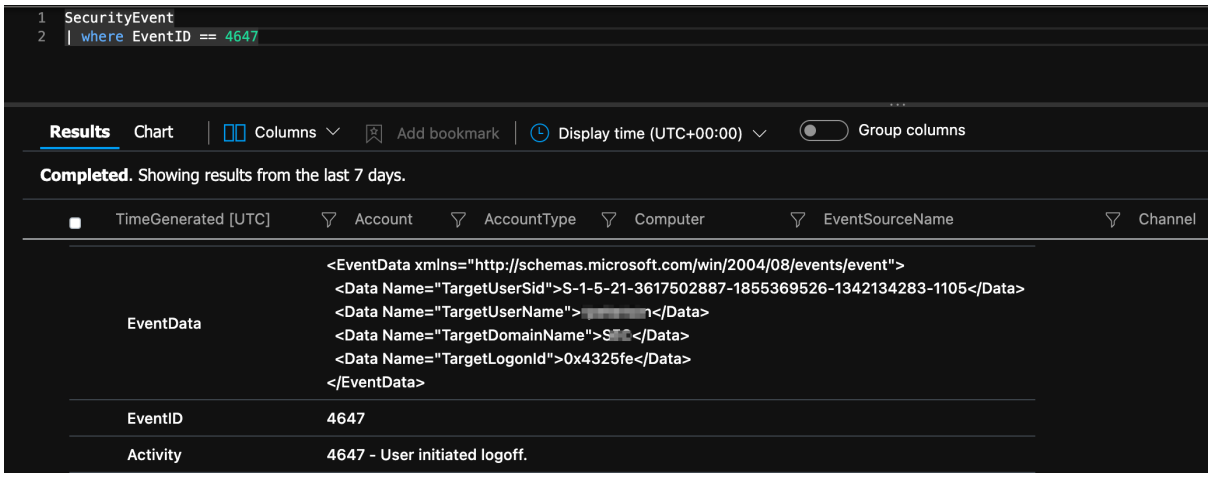
The union operator can join together more than one tables.

```
ZeekHTTP
| project TimeGenerated, Type,src_port
| union withsource=SourceZeek (ZeekDNS)
| union (ZeekSMB)
```

Parse

The *parse* operator can be used to parse data stored in XML format.

Example - in the following example we see that the data we want to process from *"Event ID 4647 - User initiated logoff"* is stored inside the *EventData* column in an XML format. The goal of this example is to extract the data stored inside the column and place it into different columns.



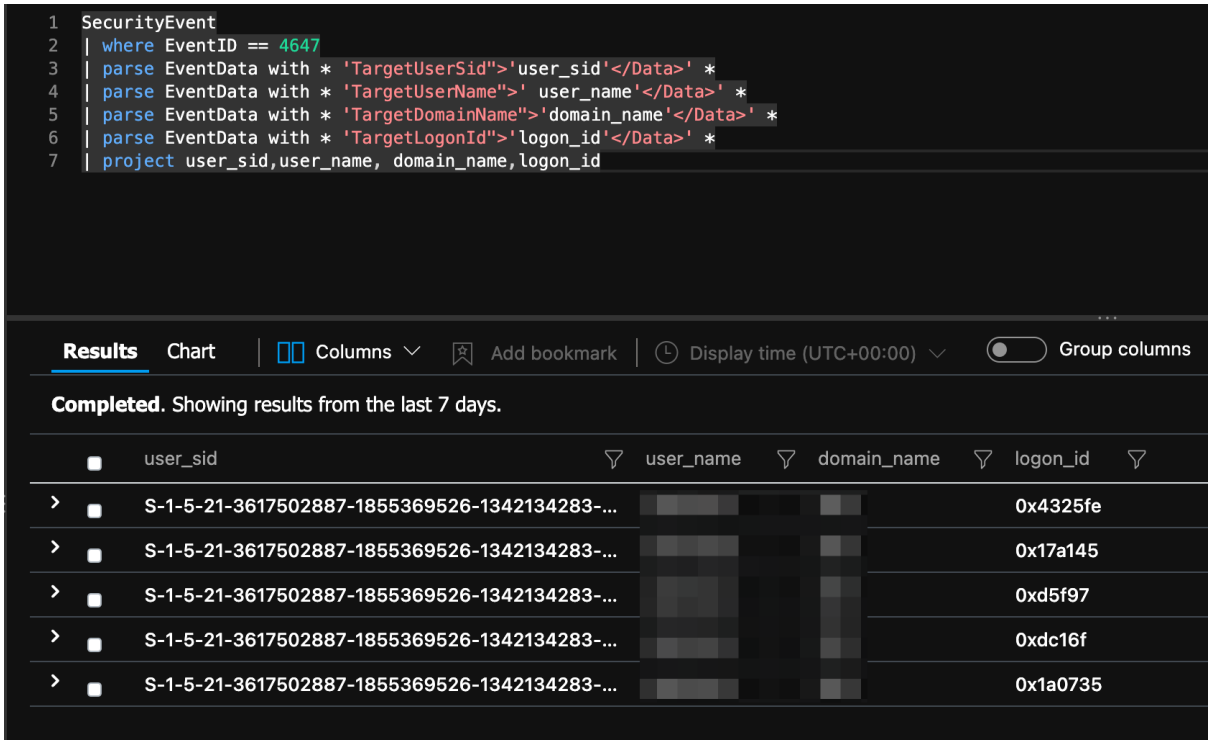
From the output we see that different headers exist

- *TargetUserSid*
- *TargetUserName*
- *TargetDomainName*
- *TargetLogonId*

- We need to tell the query from which column we want to parse from. In the example below the column is *EventData*.

- We then tell the parser to parse everything until it encounters the first column we are interested in which is the *TargetUserSid">*.
- Then we tell the parser how we want to name the new column which will hold the results of the previous command i.e the extracted *TargetUserSid*. In this example the results will go under the *user_sid* column.
- We then tell the parser to stop when it encounters the end of the line signified by *</Data>*.
- Finally the same process is repeated for all data of interest

```
SecurityEvent
| where EventID == 4647
| parse EventData with * 'TargetUserSid">'user_sid'</Data>' *
| parse EventData with * 'TargetUserName">' user_name'</Data>' *
| parse EventData with * 'TargetDomainName">'domain_name'</Data>' *
| parse EventData with * 'TargetLogonId">'logon_id'</Data>' *
| project user_sid,user_name, domain_name,logon_id
```



Similar operators are *parse_xml*, *parse_json* and *parse_csv*.

Materialize

The *materialize* function is useful in situations where we want to store the output of one query in order to be used in other subqueries without the need of re-running it as it is cached.

```
let logins = materialize (
SecurityEvent
| where EventID == 4624
| project TargetUserSid);
logins
| summarize by TargetUserSid
```

External data

The externaldata operator allows for importing data from external storage sources. Multiple formats are included but not limited to

- txt
- csv
- json

and can be defined with

```
with(format="csv")
```

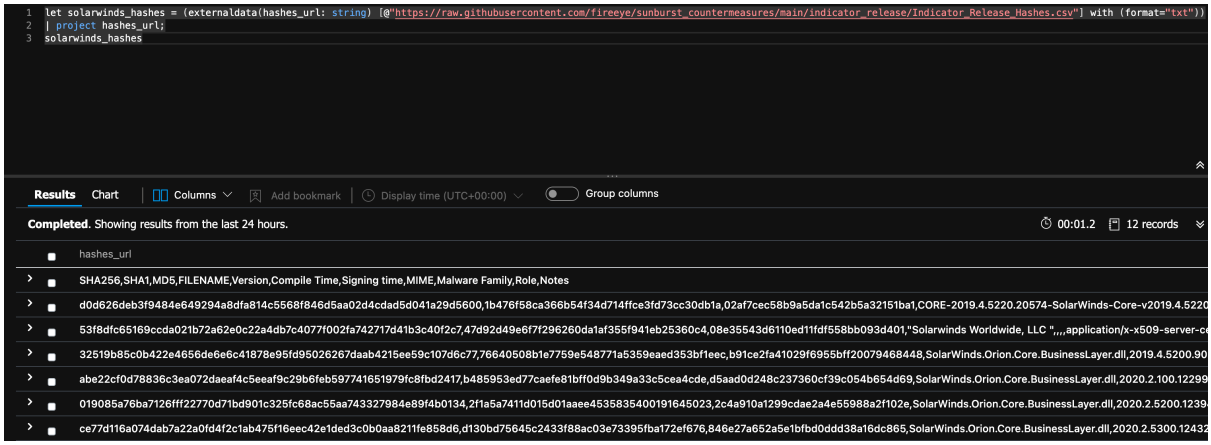
All of the supported formats are here

For this example we will use the Indicator_Release_Hashes.csv from FireEye related to the SolarWinds attack

The first step is to load the data into the *hashes_url* variable which will hold the contents of the CSV file. the variable can be named anything and we tell KQL its type, in this case a string.

```
let solarwinds_hashes = (externaldata(hashes_url: string) [@"https://raw.githubusercon
tent.com/fireeye/sunburst_countermeasures/main/indicator_release/Indicator_Release_Has
hes.csv"] with (format="txt"))
| project hashes_url;
```

We then project the *hashes_url* to load the contents of the CSV file into the *solarwinds* variable.



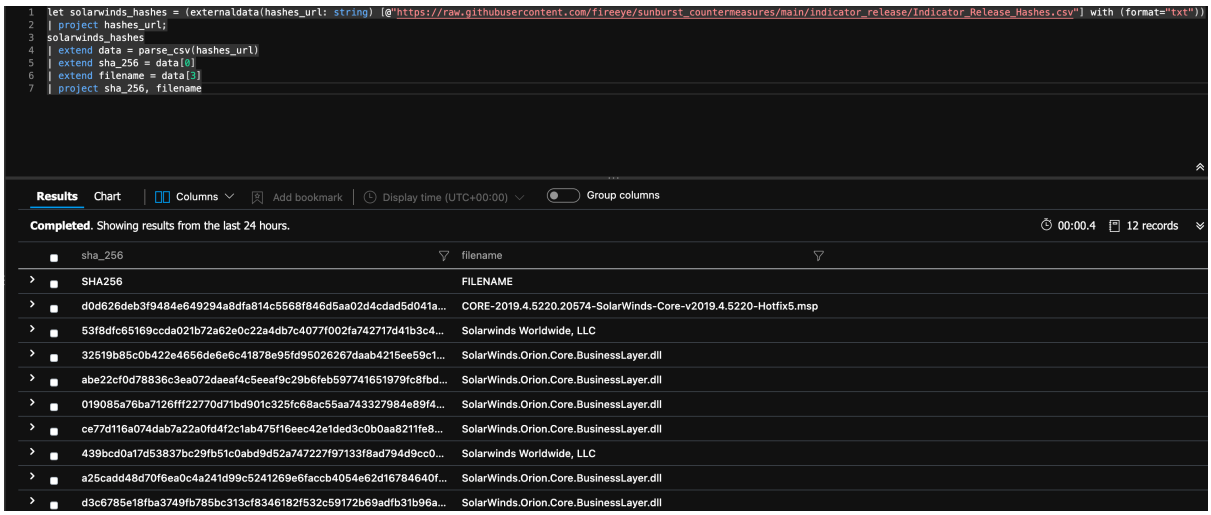
From the source we know that the data headers are divided as follows

SHA256 SHA1 MD5 FILENAME Version Compile Time Signing time MIME Malware Family Role Notes

Since the data are loaded into an array we can call specific elements of the array from 0 to 9

In this example the only columns of interest are the *SHA256* and *FILENAME* columns therefore only elements [0] and [3] are used.

```
let solarwinds_hashes = (externaldata(hashes_url: string) [@"https://raw.githubusercon
tent.com/fireeye/sunburst_countermeasures/main/indicator_release/Indicator_Release_Has
hes.csv"] with (format="txt"))
| project hashes_url;
solarwinds_hashes
| extend data = parse_csv(hashes_url)
| extend sha_256 = data[0]
| extend filename = data[3]
| project sha_256, filename
```



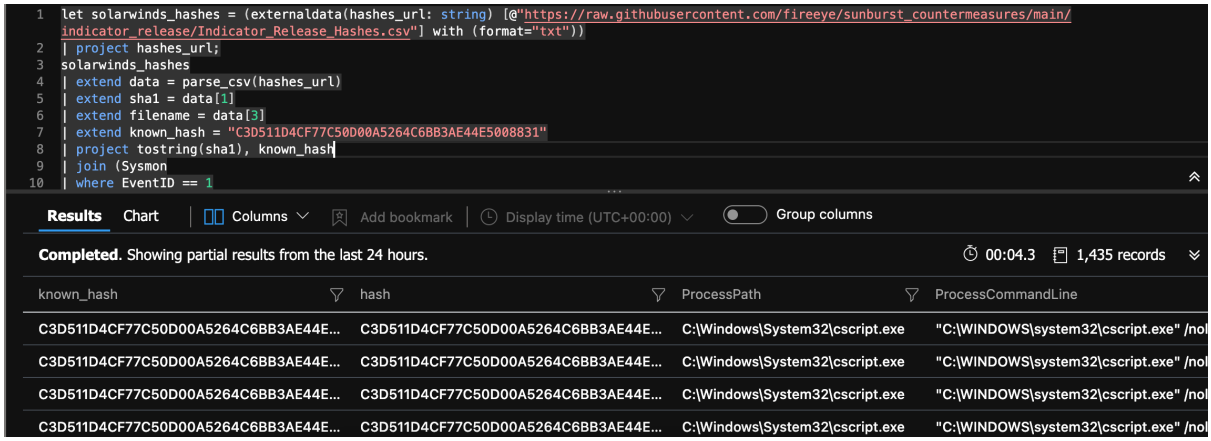
Since we now have the SHA256 of the files we can utilize this list to search for the IoCs inside the network.

Example - For this example i will use the SHA1 instead of SHA256. The SHA1 hash is kept inside *data[1]* which is the second column of the CSV file.

We know the processes created are logged by Sysmon Event ID 1 so that table will be used for joining with the data from the CSV file.

Since these IoC's do not exist in my test environment i will use a known hash of *cscript.exe* to show how the output would look like.

```
let solarwinds_hashes = (externaldata(hashes_url: string) [@"https://raw.githubusercon
tent.com/fireeye/sunburst_countermeasures/main/indicator_release/Indicator_Release_Has
hes.csv"] with (format="txt"))
| project hashes_url;
solarwinds_hashes
| extend data = parse_csv(hashes_url)
| extend sha1 = data[1]
| extend filename = data[3]
| extend known_hash = "C3D511D4CF77C50D00A5264C6BB3AE44E5008831"
| project known_hash
| join (Sysmon
| where EventID == 1
| project hash, ProcessPath, ProcessCommandLine) on $left.known_hash == $right.hash
```



Alternatively to search for all the hashes throughout the logs.

```
let solarwinds_hashes = (externaldata(hashes_url: string) [@"https://raw.githubusercon
tent.com/fireeye/sunburst_countermeasures/main/indicator_release/Indicator_Release_Has
hes.csv"] with (format="txt"))
| project hashes_url;
solarwinds_hashes
| extend data = parse_csv(hashes_url)
| extend sha1 = data[1]
| extend filename = data[3]
| project tostring(sha1)
| join (Sysmon
| where EventID == 1
| project hash, ProcessPath, ProcessCommandLine) on $left.sha1 == $right.hash
```

Render

The render operator is used to display the output of the query in various formats visually including but not limited to:

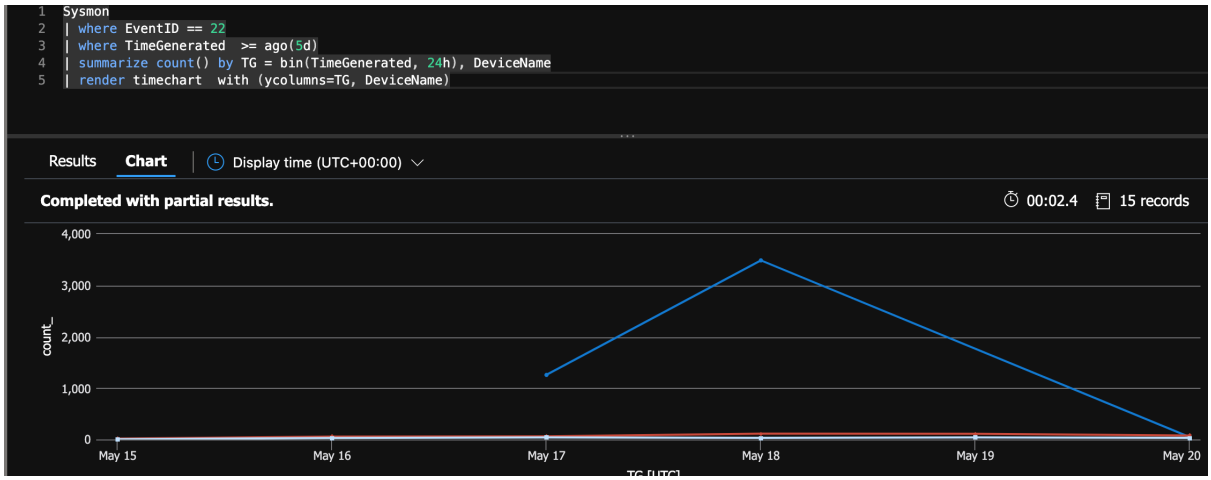
- barchart
- piechart
- columnchart
- areachart

> ⚠ The *render* operator must be the last operator in the query

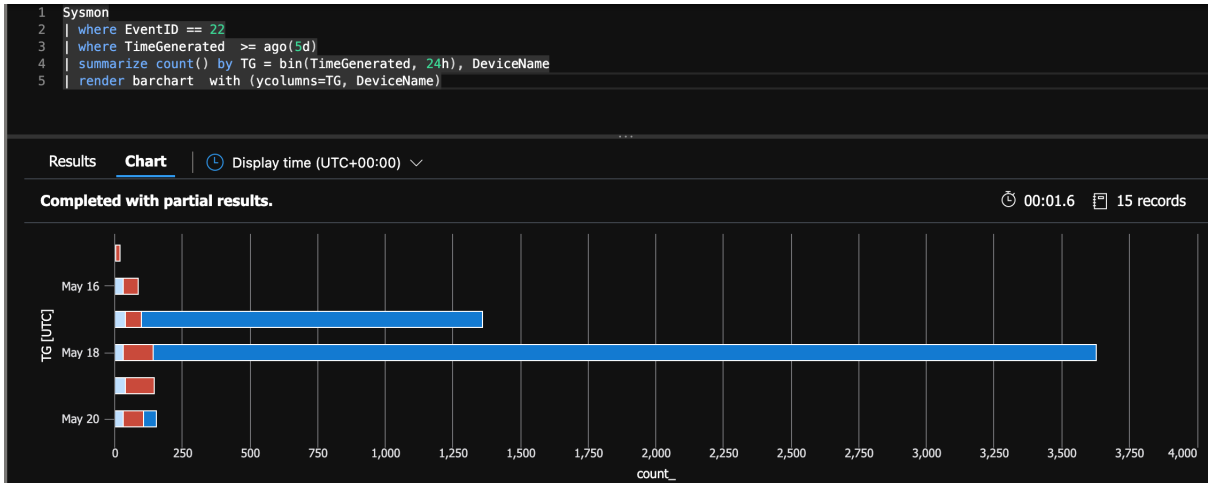
```
Sysmon
| where EventID == 1
| where TimeGenerated  >= ago(5d)
| where ProcessCommandLine has "cmd.exe"
| summarize count() by TG = bin(TimeGenerated, 24h), DeviceName
| render columnchart with (ycolumns=TG,DeviceName)
```

Depending on the goal of the query different charts can be used.

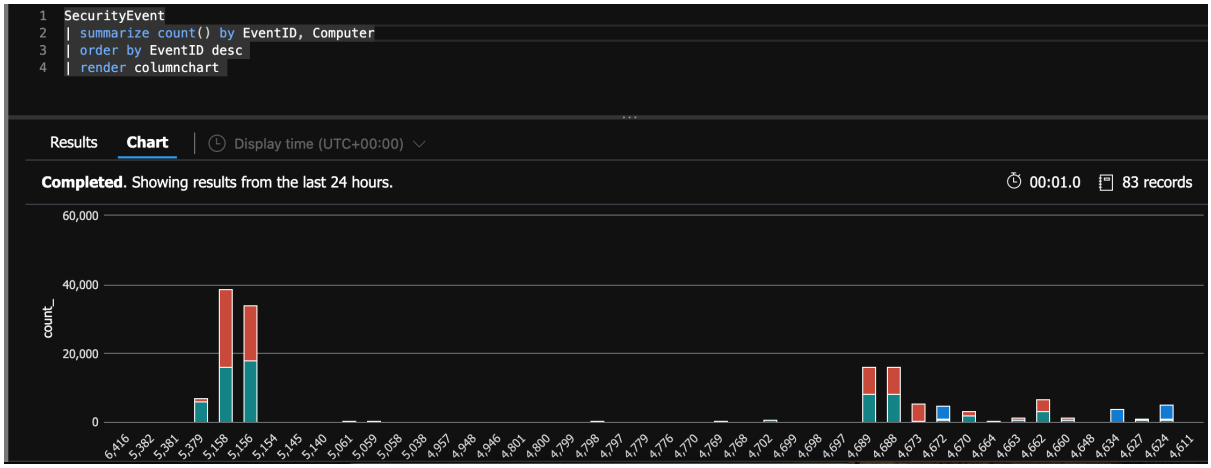
If we are trying to spot anomalous numbers of DNS requests the *timechart* might be more useful in terms of visualization.



The same query may be visualized differently.

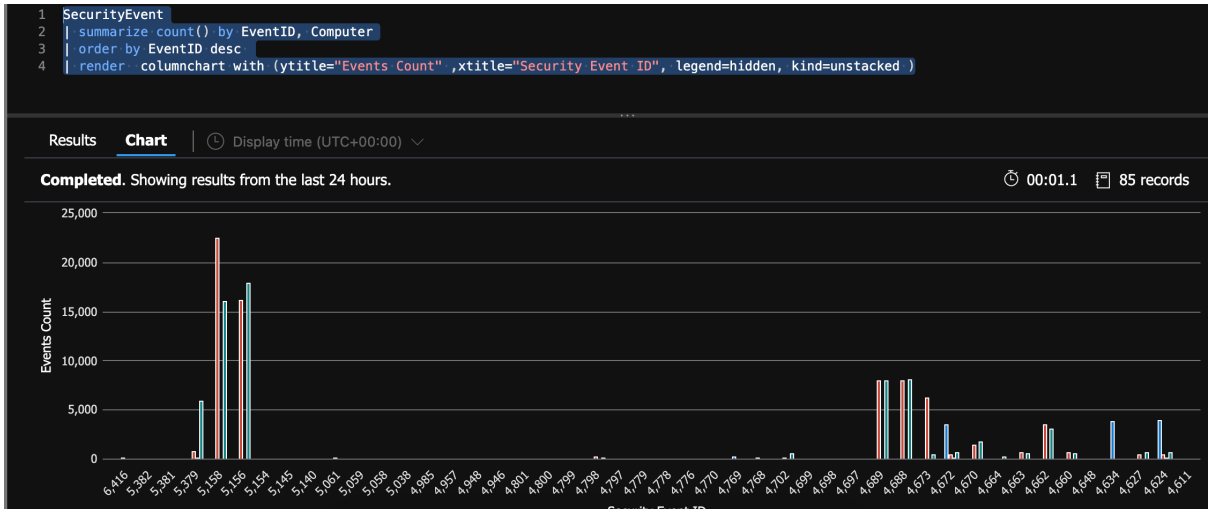


And to illustrate how a columnchart would look like.



KQL offers some granularity in terms of the elements of the visualisation through various parameters.

```
SecurityEvent
| summarize count() by EventID, Computer
| order by EventID desc
| render  columnchart with (ytitle="Events Count" ,xtitle="Security Event ID", legend=
hidden, kind=unstacked )
```



Serialize

The *serialize* operator is very important to be able to use certain functions. This operator converts the output to a serialized(ordered) output so that functions such as Window Functions can be applied on the data. Window functions include

- next()
- prev()
- row_checksum()
- row_number()
- row_rank()
- row_window_session()

The output from the following operators is serialized therefore there is not need to apply the *serialize* operator again.

- range
- sort
- order
- top-hitters
- ...

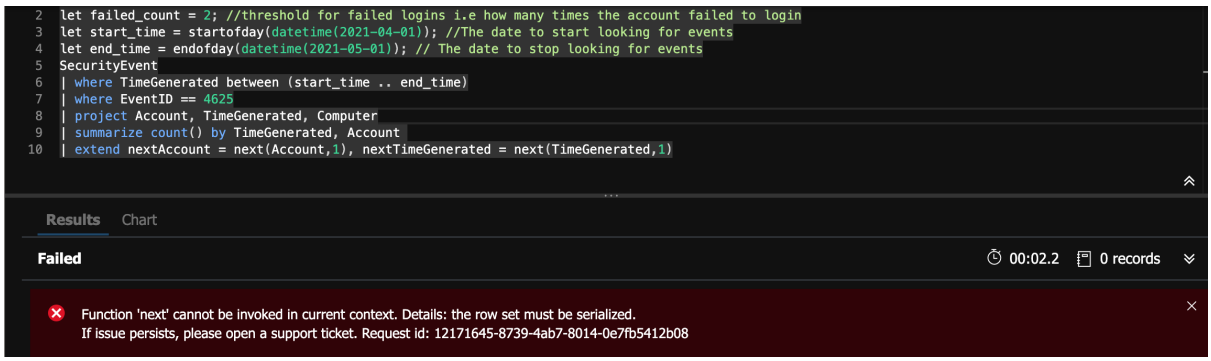
The following operators output is not serialized therefore it needs to be applied:

- sample
- sample-distinct
- join
- top-nested
- summarize
- make-series
- ...

Example - Attempting to run a window function on non-serialized data.

```
let start_time = startofday(datetime(2021-04-01)); //The date to start looking for eve
nts
let end_time = endofday(datetime(2021-05-01)); // The date to stop looking for events
SecurityEvent
| where TimeGenerated between (start_time .. end_time)
| where EventID == 4625
| project Account, TimeGenerated, Computer
| summarize count() by TimeGenerated, Account
| extend nextAccount = next(Account,1), nextTimeGenerated = next(TimeGenerated,1)
```

In the example above it was attempted to use the *next* function on non-serialized output from the *summarize* operator and KQL complains about it.




```
let start_time = startofday(datetime(2021-04-01)); //The date to start looking for eve
nts
let end_time = endofday(datetime(2021-05-01)); // The date to stop looking for events
SecurityEvent
```

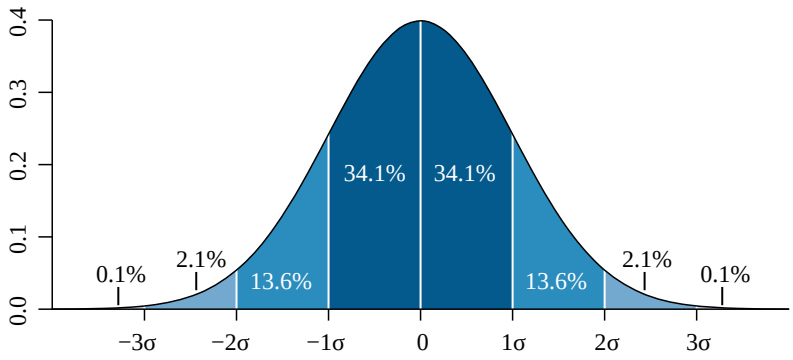
```
| where TimeGenerated between (start_time .. end_time)
| where EventID == 4625
| project Account, TimeGenerated, Computer
| summarize count() by TimeGenerated, Account
| serialize
| extend nextAccount = next(Account,1), nextTimeGenerated = next(TimeGenerated,1)
```

The same query but using the *serialize* operator on the output data fixes the problem. Alternatively another operator could be used which outputs serialized data such as the *sort* operator.

```
let start_time = startofday(datetime(2021-04-01)); //The date to start looking for eve
nts
let end_time = endofday(datetime(2021-05-01)); // The date to stop looking for events
SecurityEvent
| where TimeGenerated between (start_time .. end_time)
| where EventID == 4625
| project Account, TimeGenerated, Computer
| sort by TimeGenerated asc, Account
| extend nextAccount = next(Account,1), nextTimeGenerated = next(TimeGenerated,1)
```

Stdev

The *stdev* (standard deviation) operator assists in measuring the amount of dispersion in a set of values. The bell shaped curve displayed below shows the normal distribution where each band has 1 standard deviation.



A low standard deviation indicates that the values tend to be close to the mean of the set, while a high standard deviation indicates that the values are spread out over a wider range.

The following is a great example by Binary Defense to track down C2 beacon connections.

```
let starttime = 48h;
let endtime = 1m;
let TimeDeltaThreshold = 2;
let TotalEventsThreshold = 15;
let DurationThreshold = 900;
let StandardDeviationThreshold = 100;
Sysmon
| where EventID==3
| where TimeGenerated between (ago(starttime)..ago(endtime))
| project TimeGenerated, Computer, process_path, src_ip, src_port, dst_ip, dst_port
| sort by src_ip asc, dst_ip asc, TimeGenerated asc
| serialize
| extend nextTimeGenerated = next(TimeGenerated, 1), nextDeviceId = next(Computer, 1),
nextDstIP = next(dst_ip, 1)
| extend TimeDeltaInSeconds = datetime_diff("second", nextTimeGenerated, TimeGenerate
d) // compute time difference between subsequent connections
| where Computer == nextDeviceId and nextDstIP == dst_ip
| where TimeDeltaInSeconds > TimeDeltaThreshold
| project TimeGenerated, TimeDeltaInSeconds, Computer, process_path, src_ip, src_port,
dst_ip, dst_port
| summarize avg(TimeDeltaInSeconds), count(), min(TimeGenerated), max(TimeGenerated),
Duration=datetime_diff("second", max(TimeGenerated), min(TimeGenerated)),
StandardDeviation=stdev(TimeDeltaInSeconds), TimeDeltaList=make_list(TimeDeltaInSecond
s) by Computer, src_ip, dst_ip, process_path
| where count_ > TotalEventsThreshold
| where StandardDeviation < StandardDeviationThreshold
| where Duration >= DurationThreshold
| order by StandardDeviation asc
```

Walkthrough of the query

```
let starttime = 48h;
let endtime = 1m;
let TimeDeltaThreshold = 2;
let TotalEventsThreshold = 15;
let DurationThreshold = 900;
let StandardDeviationThreshold = 100;
Sysmon
| where EventID==3
| where TimeGenerated between (ago(starttime)..ago(endtime))
| project TimeGenerated, Computer, process_path, src_ip, src_port, dst_ip, dst_port
| sort by src_ip asc, dst_ip asc, TimeGenerated asc
```

The query begins by first finding all Sysmon EventID 3 which are in the time period defined. Other than the starttime and endtime the result of the defined constants with let are used later on.

```
| serialize
| extend nextTimeGenerated = next(TimeGenerated, 1), nextDeviceId = next(Computer, 1),
nextDstIP = next(dst_ip, 1)
| extend TimeDeltaInSeconds = datetime_diff("second", nextTimeGenerated, TimeGenerate
d)
```

The output is then serialized so that the *next* operator can be used. The *next* operator is user to put the immediate next row's *TimeGenerated, Computer* and *dst_ip* into new variables. Lastly the difference in seconds between the current row and the next row is calculated and entered in the *TimeDeltaInSeconds* variable.

```
| where Computer == nextDeviceId and nextDstIP == dst_ip
| where TimeDeltaInSeconds > TimeDeltaThreshold
| project TimeGenerated, TimeDeltaInSeconds, Computer, process_path, src_ip, src_port,
dst_ip, dst_port
```

The following line in the query is to check if the current row's *Computer* and *dst_ip* are the same with the next row so that similar lines are compared. If that condition is satisfied the query checks next that the *TimeDeltaInSeconds* calculated before is larger that the threshold defined in the *TimeDeltaThreshold* constant. The *TimeDeltaThreshold* is defined as 2 seconds therefore the query is excluding all events where the threshold is less than 2 seconds so very close events are not taken into consideration.

```
| summarize avg(TimeDeltaInSeconds), count(), min(TimeGenerated), max(TimeGenerated),
Duration=datetime_diff("second", max(TimeGenerated), min(TimeGenerated)),
StandardDeviation=stdev(TimeDeltaInSeconds), TimeDeltaList=make_list(TimeDeltaInSecond
s) by Computer, src_ip, dst_ip, process_path
```

The query then calculates the average TimeDeltaInSeconds and the first and last events recorded based on the TimeGenerated. This is then used to calculated the duration between the first and last event is seconds.

The standard deviation is calculated based on the TimeDeltaInSeconds i.e the time difference between each record (connection). As C2 beaconing is usually at predefined intervals a low standard deviation will show that the values are very close to each other therefore beaconing could possibly be taking place.

Lastly a list is made containing the *TimeDeltaInSeconds* which is used for presentation purposes mostly.

```
| where count_ > TotalEventsThreshold
| where StandardDeviation < StandardDeviationThreshold
| where Duration >= DurationThreshold
| order by StandardDeviation asc
```

Lastly the query checks that the count of the series of events is more than 15 and that the threshold for the standard deviation is less than 100.

Another example of the *stdev* operator and also the *variance* operator this time looking at the failed logins of users.

```
let failed_threshold = 5; //threshold to use for failed login times i.e how much time
 between each failed login
let failed_count = 2; //threshold for failed logins i.e how many times the account fai
led to login
let stdev_threshold = 1;
let start_time = startofday(datetime(2021-04-01)); //The date to start looking for eve
nts
let end_time = endofday(datetime(2021-05-01)); // The date to stop looking for events
SecurityEvent
| where TimeGenerated between (start_time .. end_time)
| where EventID == 4625
| 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, TimeGenerate
d)
| where  TimeDeltaInSeconds <= failed_threshold
| project TimeGenerated, TimeDeltaInSeconds, Account, Computer
| summarize Failed_Logins = count(),avg(TimeDeltaInSeconds),first_failed = min(TimeGen
erated), last_failed = max(TimeGenerated), standarddev = stdev(TimeDeltaInSeconds),var
iance= variance(TimeDeltaInSeconds), TimeDeltaList=make_list(TimeDeltaInSeconds) by Ac
count
| where standarddev < stdev_threshold
| where Failed_Logins >= failed_count
| sort by Failed_Logins desc
```

String Operators

KQL offers a variety of query operators for searching string data types. It is important to note that KQL indexes all columns of string type. When using operators such as *has, !has, hasprefix* and *!hasprefix* instead of doing a plain substring match these operators match terms.

> 💡 If the query is smaller than 3 characters or uses the contains operator the query will search in both the term index and the values in the column making the query much more resource intensive

The complete list of string operators can be found here

Some important operators include:

- ==
- has
- !has
- haspreffix
- hassuffix
- contains
- startswith
- endswith
- in~
- in
- !~

KQL Performance Tricks

- instead of `=~`, use `==`
- instead of `in~`, use `in`
- instead of `contains`, use `contains_cs`

Lets say we want to look in Event ID 1 of Sysmon to search for any events where the username of the user generating the event is demo_user. This can be achieved either with the *has* operator as well as the *hassuffix* operator as the usernames are in the format of DOMAIN\USERNAME.

```
Sysmon
| where TimeGenerated >= ago(3h)
| where EventID ==1
| where AccountName  has "demo_user"
```

```
Sysmon
| where TimeGenerated >= ago(3h)
| where EventID ==1
| where AccountName  hassuffix "system"
```

If we are searching for a string in the beginning of the work the *hasprefix* operator can be used.

```
Sysmon
| where TimeGenerated >= ago(3h)
| where EventID ==1
| where DeviceName  hasprefix "work"
```

Similarly the *startswith* and *endswith* operators can be used.

Let's say we are looking to identify all processes running with high integrity level. The same result can be achieved in multiple ways.

Since we don't know about the string case the ~ is used.

```
Sysmon
| where TimeGenerated >= ago(30h)
| where EventID == 1
| where ProcessIntegrityLevel  in~ ("high")
```

Or using a negated approach to exclude everything except the one integrity level we are interested in.

```
Sysmon
| where TimeGenerated >= ago(3h)
| where EventID == 1
| where ProcessIntegrityLevel  !in~ ("system" , "medium")
```

```
Sysmon
| where TimeGenerated >= ago(3h)
| where EventID == 1
| where ProcessIntegrityLevel == "High"
```

```
Sysmon
| where TimeGenerated >= ago(3h)
| where EventID ==1
| where ProcessIntegrityLevel =~ "high"
```

```
Sysmon
| where TimeGenerated >= ago(3h)
| where EventID == 1
| where ProcessIntegrityLevel has "high"
```

Because of the case sensitivity the below query will not return any results

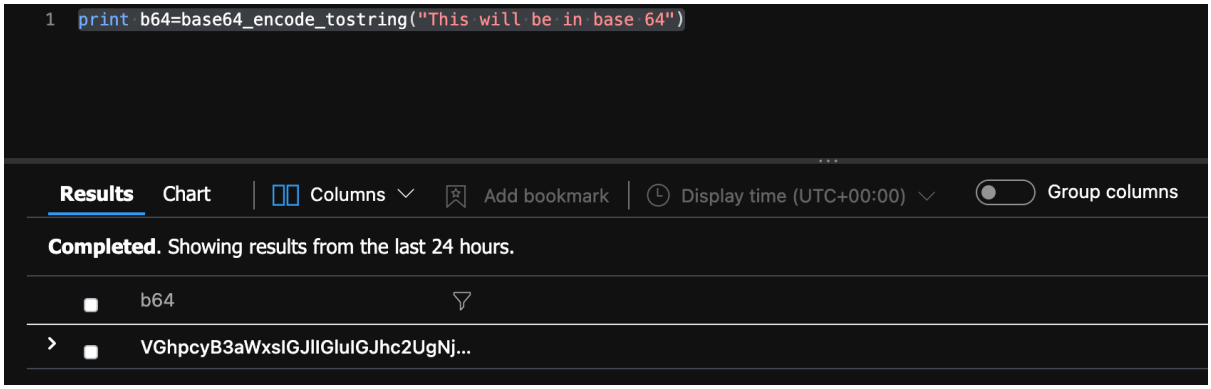
```
Sysmon
| where TimeGenerated >= ago(3h)
| where EventID == 1
| where ProcessIntegrityLevel has_cs "high"
```

Scalar Functions

Base64 Encode/Decode

This function can be used to perform base64 operations

```
print b64=base64_encode_tostring("This will be in base64")
```



Similarly the same functionality can be used to decode a base64 string

```
print plaintext=base64_decode_tostring("VGhpcyB3aWxsIGJlIGluIGJhc2U2NA==")
```

Example - Hunting for PowerShell base64 encoded commands

A PowerShell encoded command will look something like the below

```
powershell -encodedCommand cABzAz==
```

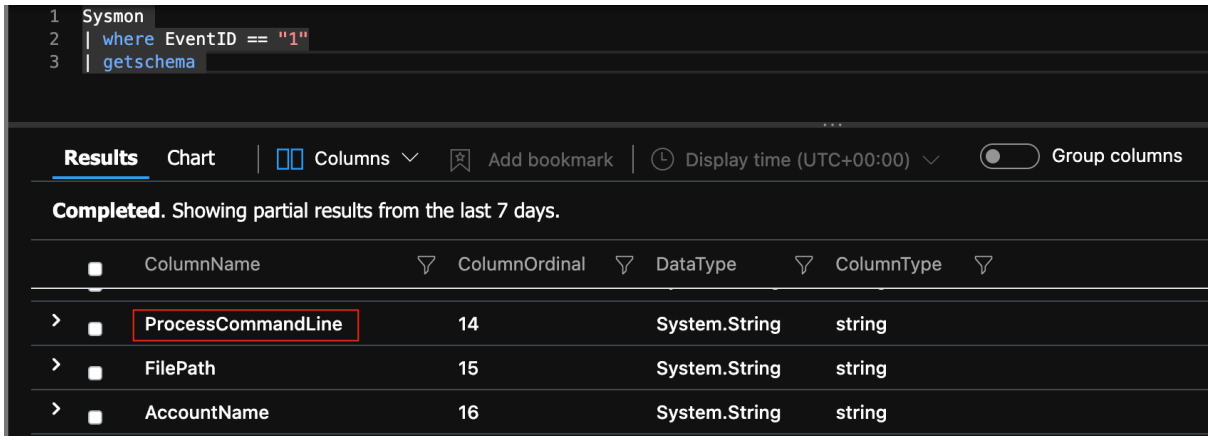
The *-encoded* command part is needed for PowerShell to know that a base64 string is next. Nonetheless the following will also work so they need to be kept in mind when designing the query.

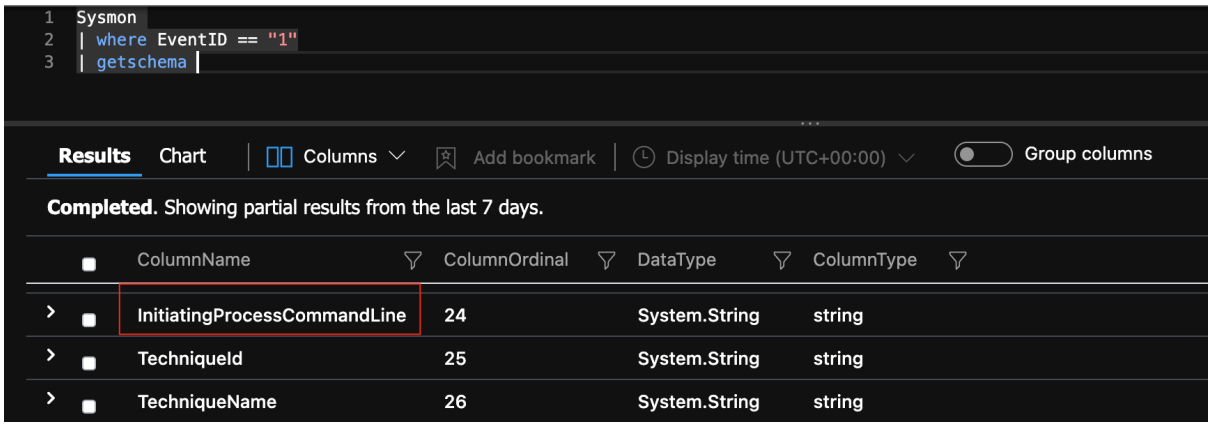
- -encodedcommand
- -enc
- -e

We know that PowerShell will create a new process therefore EventID 1 is the relevant Sysmon event we should be looking into.

The first step is to check the schema of EventID 1 and understand which fields may be of use.

Two fields seem to be the ones that we should be looking into which refer to the actual command line that was executed.





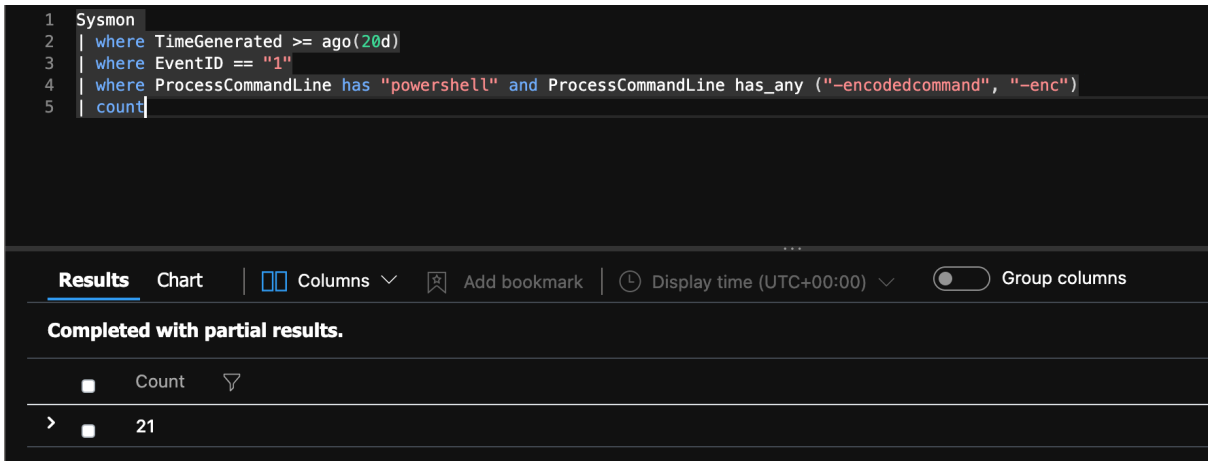
Next we can start work on the query

First we need to focus on the process command line that contains powershell.

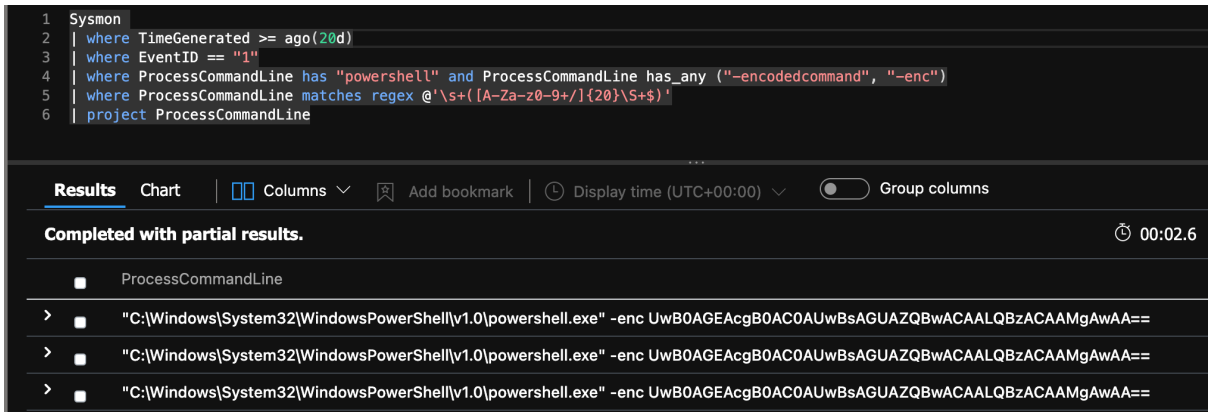
```
Sysmon
| where TimeGenerated >= ago(2d)
| where EventID == "1"
| where ProcessCommandLine has "powershell"
```

Next we need to combine the above with the known strings for encoded commands. In this example we will be searching only for -enc and -encodedcommand

```
Sysmon
| where TimeGenerated >= ago(2d)
| where EventID == "1"
| where ProcessCommandLine has "powershell" and ProcessCommandLine has_any ("-encodedc
ommand", "-enc")
```



The next addition to the query is to ensure that it contains a base64 string. This can be achieved with regex and the *matches_regex* string operator.



Next we need to extract that string in order to decode it.

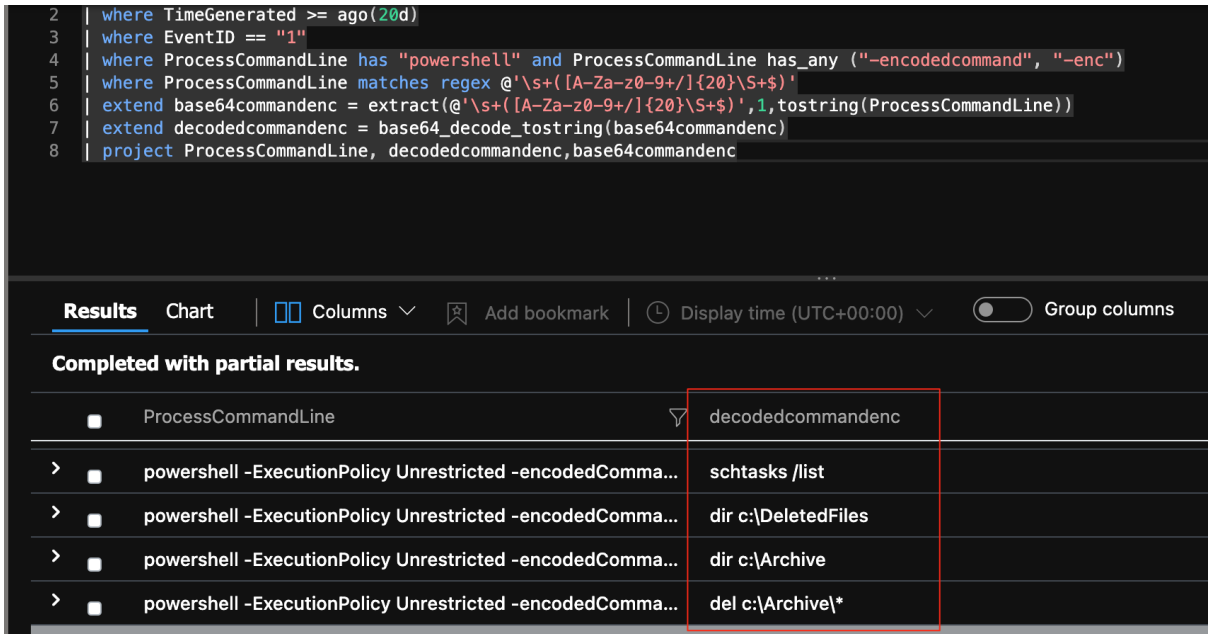
We are creating a new variable to hold the result of the extracted string using the *extract* operator.

The *extract* operator takes as input the string to match, the position to return and the input that the extraction should take place on.

```
Sysmon
| where TimeGenerated >= ago(20d)
| where EventID == "1"
| where ProcessCommandLine has "powershell" and ProcessCommandLine has_any ("-encodedc
ommand", "-enc")
| where ProcessCommandLine matches regex @'\s+([A-Za-z0-9+/]{20}\S+$)'
| extend base64commandenc = extract(@'\s+([A-Za-z0-9+/]{20}\S+$)',1,tostring(ProcessCo
mmandLine))
```

In the last step the *base64_decode_tostring* operator is used to convert the base64 string to readable text.

```
Sysmon
| where TimeGenerated >= ago(2d)
| where EventID == "1"
| where ProcessCommandLine has "powershell" and ProcessCommandLine has_any ("-encodedc
ommand", "-enc")
| where ProcessCommandLine matches regex @'\s+([A-Za-z0-9+/]{20}\S+$)'
| extend base64commandenc = extract(@'\s+([A-Za-z0-9+/]{20}\S+$)',1,tostring(ProcessCo
mmandLine))
| extend decodedcommandenc = base64_decode_tostring(base64commandenc)
| project ProcessCommandLine, decodedcommandenc
```



A slightly modified version to use other functions as well:

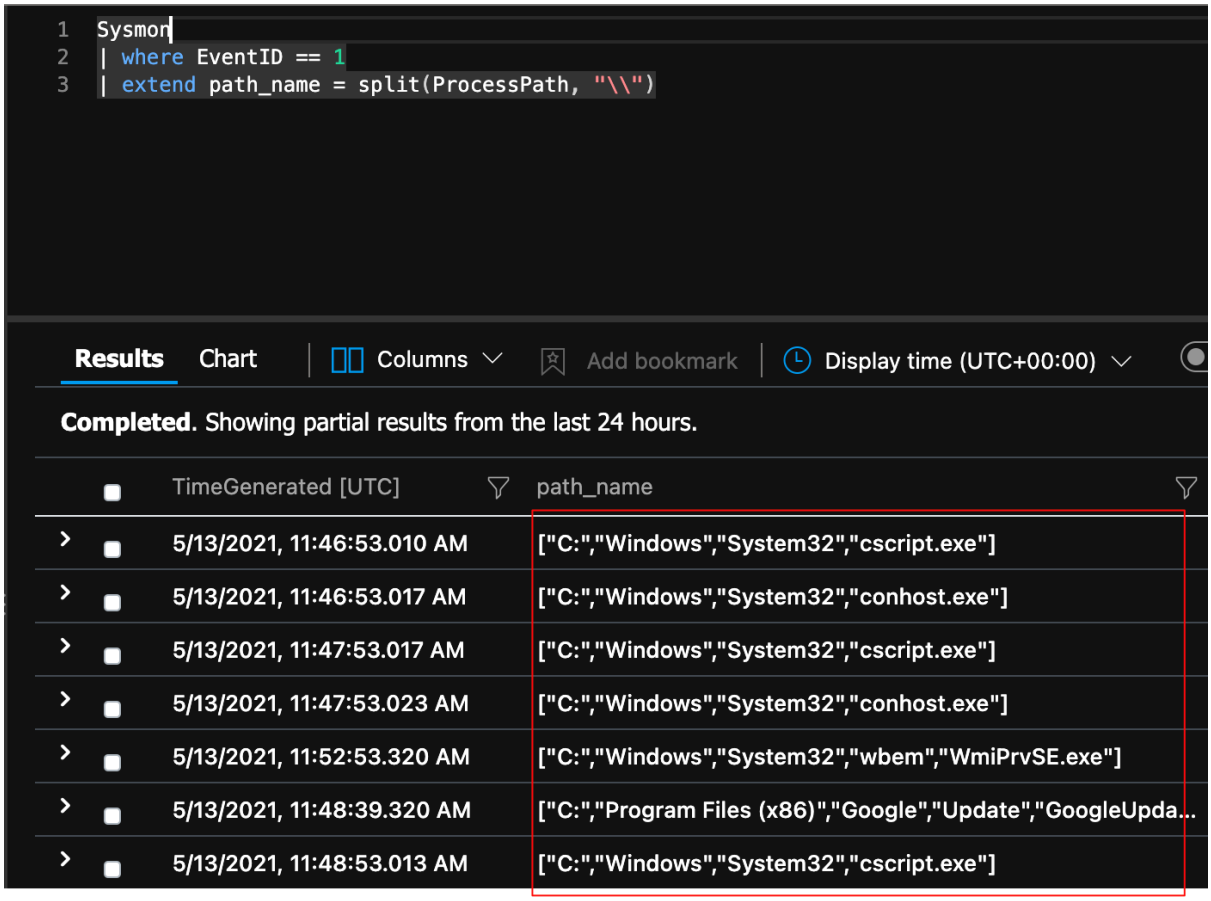
```
| where TimeGenerated >= ago(20d)
| where EventID == "1"
| where FileName has "powershell" and ProcessCommandLine has_any ("-encodedcommand",
 "-enc")
| where ProcessCommandLine matches regex @'\s+([A-Za-z0-9+/]{20}\S+$)'
| extend Decoded_command = iif(ProcessCommandLine matches regex @'\s+([A-Za-z0-9+/]{2
0}\S+$)', decodedcommandenc = base64_decode_tostring(extract(@'\s+([A-Za-z0-9+/]{20}\S
+$)',1,tostring(ProcessCommandLine))),"Error Decoding")
| project TimeGenerated, ProcessCommandLine, Decoded_command,DeviceName, AccountName
```

⚠ The method above is not foolproof and there are still ways to bypass this detection

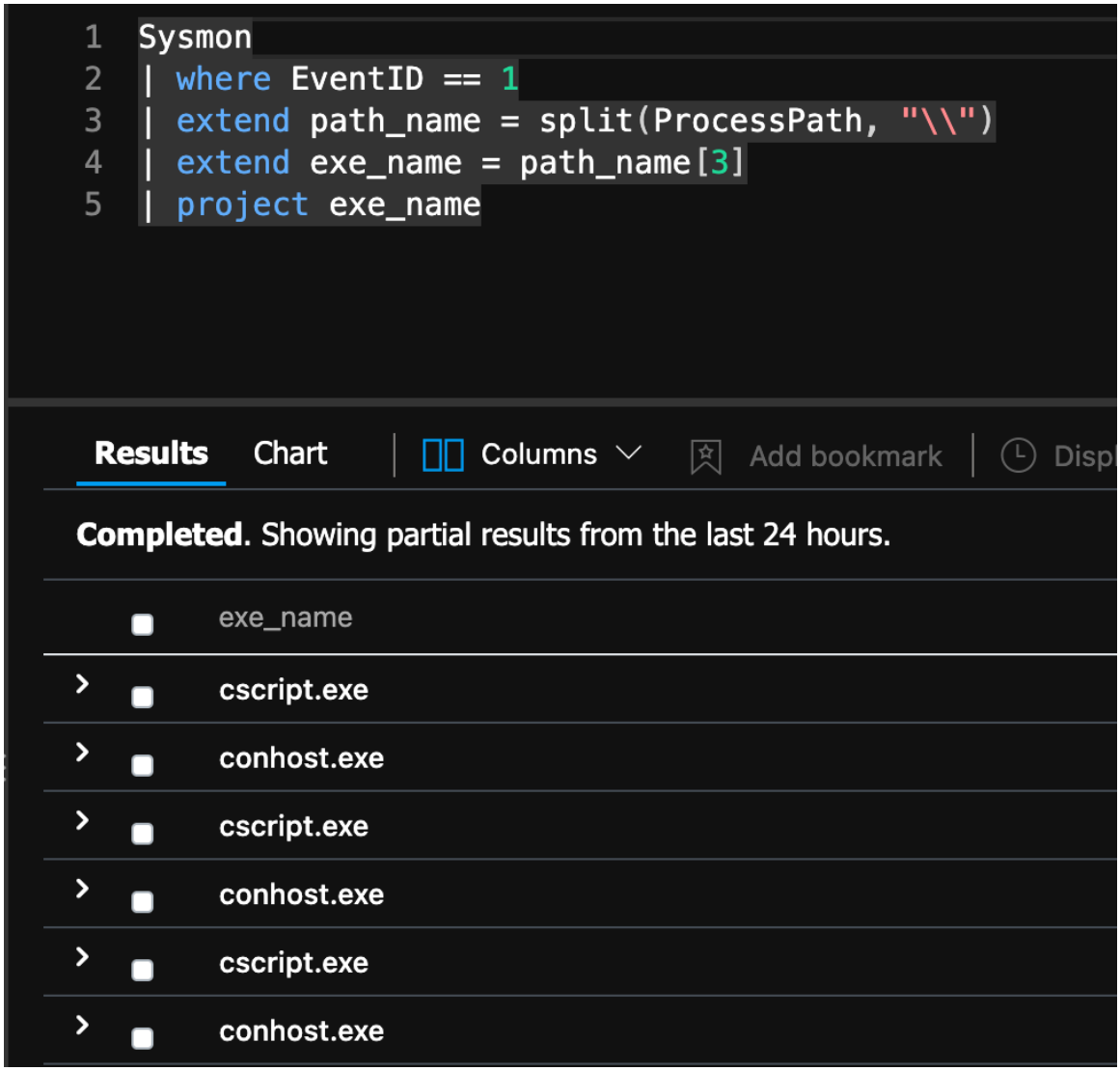
Split

The *split* function divides the input string according to the delimiter provided and returns a string array with the contained substrings.

```
Sysmon
| where EventID == 1
| extend path_name = split(ProcessPath, "\\")
```



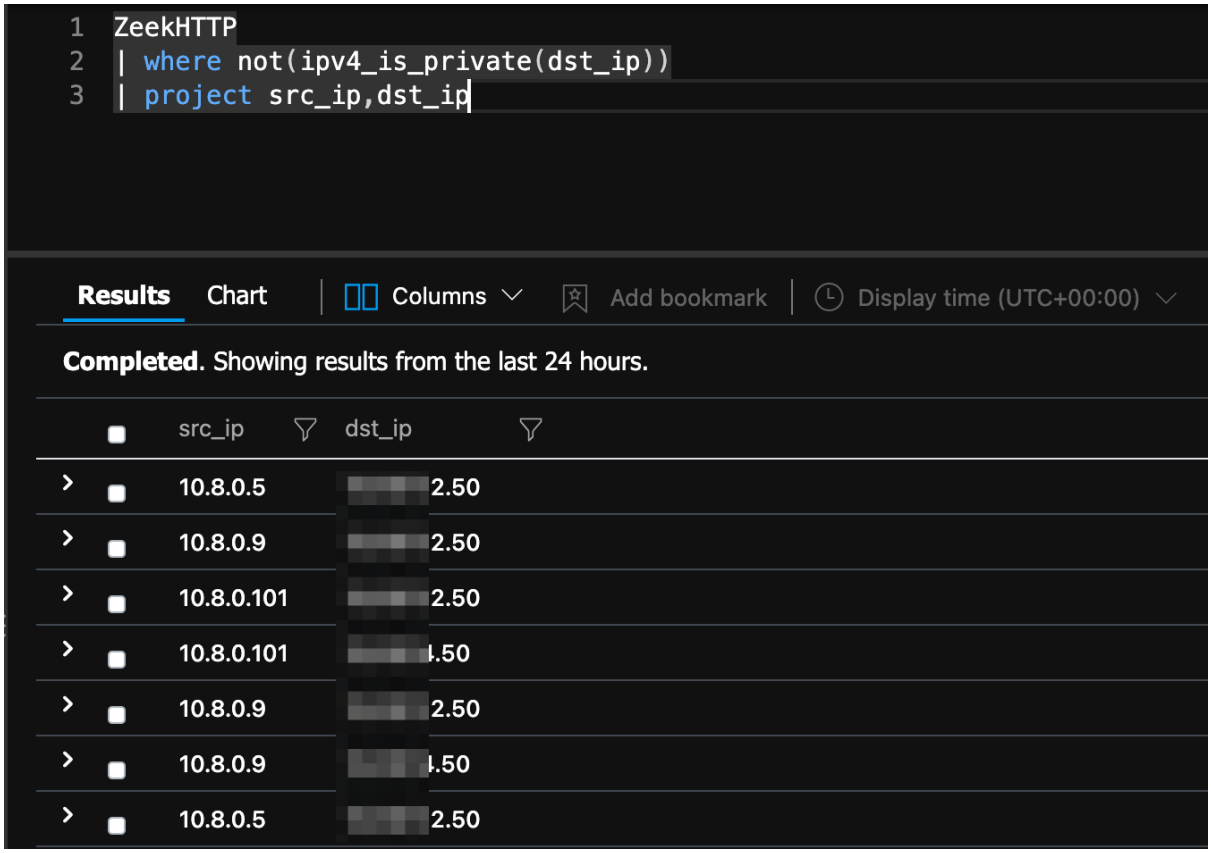
In the example above the *path_name* column contains the split command line based on the delimiter "\". Because the delimiter needs to be escaped an additional "\" is used. We can then address specific elements of the array, in this example the 4th element which contains the executable name.



ipv4_is_private

The *ipv4_is_private* operator checks if IPv4 string address belongs to a set of private network IPs i.e part of the following IP ranges.

- 10.0.0.0 – 10.255.255.255
- 172.16.0.0 – 172.31.255.255
- 192.168.0.0 – 192.168.255.255



In this example we are quering the Zeek HTTP logs to identify any outgoing connections to not private IP addresses i.e to the Internet. First we identify the private ipv4 IPs and use the *not* operator to reverse the boolean operation.

ipv4_is_match

In a similar manner the *ipv4_is_match* operator can be used to include or exclude IPs

Let's say we want to analyse the DNS requests made but also excluding results where the *dst_ip* matches Google's IP. This can be achieved with the following query that we use the *dst_ip* column as input to be compared against the string 8.8.8.8.

```
ZeekDNS
| where not(ipv4_is_match(dst_ip,"8.8.8.8"))
```

Entire ranges can also be used in CIDR format.

```
ZeekDNS
| where not(ipv4_is_match(dst_ip,"8.8.8.8")) and not(ipv4_is_match(dst_ip,"10.10.10.1/
24"))
```

Iif

The *iif* operator evaluates the first argument and returns the second argument if true or the third if false.

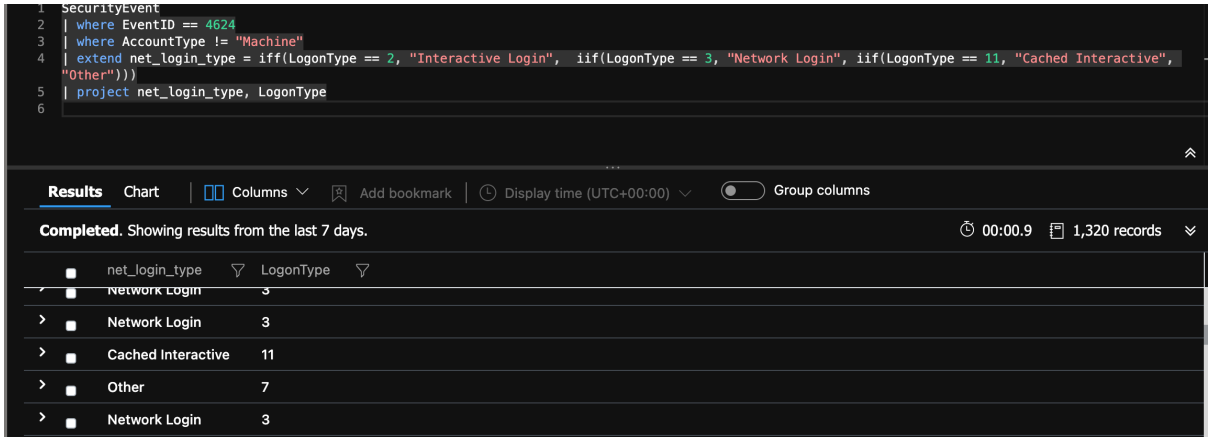
```
SecurityEvent
| where EventID == 4624
| where AccountType != "Machine"
| extend admin_hunting = iff(Account has_any("admin", "administrator"), "Admin Found",
"Non Admin")
| project admin_hunting, Account
```

The query above will create a new column called *admin_hunting* and place "Admin Found" or "Non Admin" depending if the *Account* column has the word admin or administrator in them.

These statements can also be nested

```
SecurityEvent
| where EventID == 4624
| where AccountType != "Machine"
| extend net_login_type = iff(LogonType == 2, "Interactive Login",  iif(LogonType ==
 3, "Network Login", iif(LogonType == 11, "Cached Interactive", "Other")))
| project net_login_type, LogonType
```

What happens in this query is that the third argument which is the false result is a new *iff* query nested inside the previous one.



Isnotempty

The *isnotempty* operator is very useful to include or exclude rows from the output. The result of this operator is a boolean true or false.

```
Sysmon
| where EventID == 1
| where isnotempty(AccountName)
```

This query will return only the rows where the *AccountName* is not empty.

The query can be negated so only the empty values will be returned.

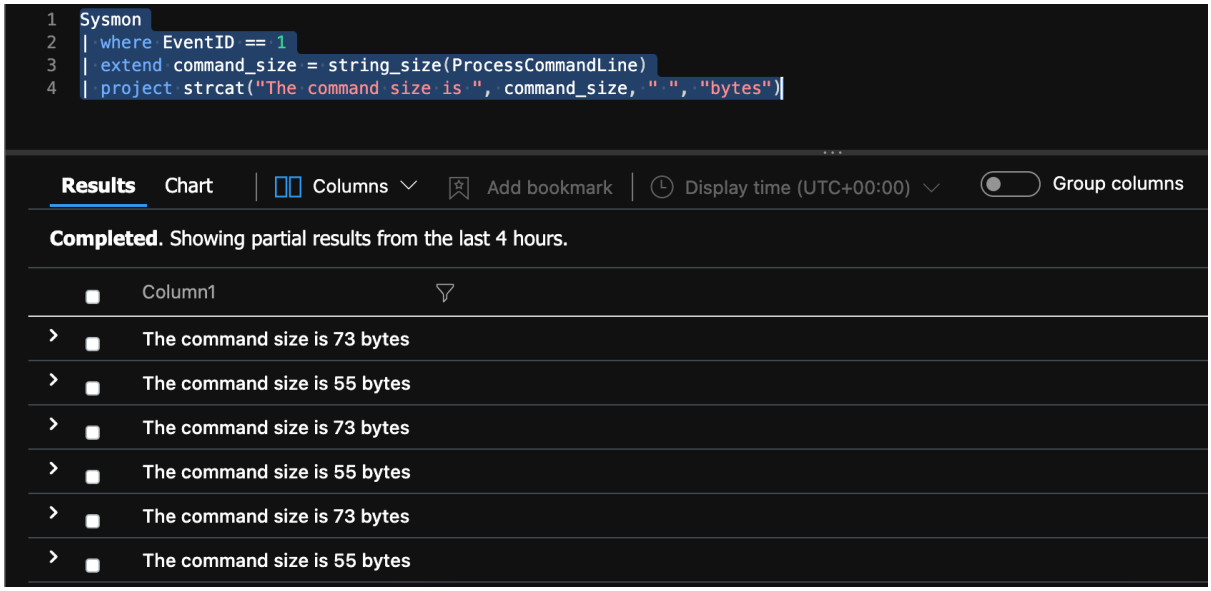
```
Sysmon
| where EventID == 1
| where not(isnotempty(DeviceName))
```

String_size

The *string_size* operator simply returns the size of the input string as bytes.

```
Sysmon
| where EventID == 1
| extend command_size = string_size(ProcessCommandLine)
| project strcat("The command size is ", command_size, " ", "bytes")
```

In combination with the *strcat* operator which concatenates strings the output is as follows.



Tolower

The *tolower* function converts the input string to lowercase.

```
Sysmon
| where EventID == 1
| where not(tolower(ProcessPath) has_any (tolower(@"C:\Windows"), tolower(@"c:\program
files"),tolower(@"c:\programdata")))
| project ProcessPath
```

In the example above the *ProcessPath* is converted to lowercase to avoid the capitalisation issues such as C:\ vs c:\ etc. The *not* function is also used to reverse the result of the boolean query. In essence the query will return the paths of executables ran from outside the specific folders.

Toupper

Similarly the *toupper* function can be used to convert the input to uppercase.

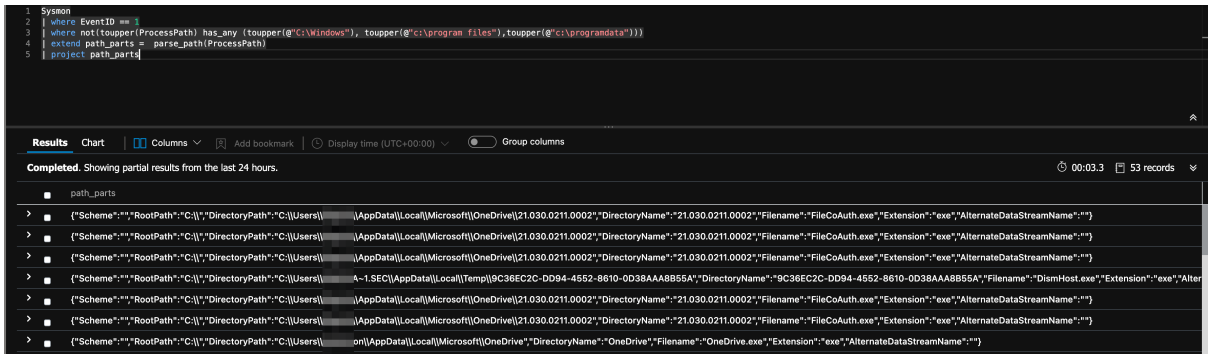
```
Sysmon
| where EventID == 1
| where not(toupper(ProcessPath) has_any (toupper(@"C:\Windows"), toupper(@"c:\program
files"),toupper(@"c:\programdata")))
| project ProcessPath
```

Parse_Path

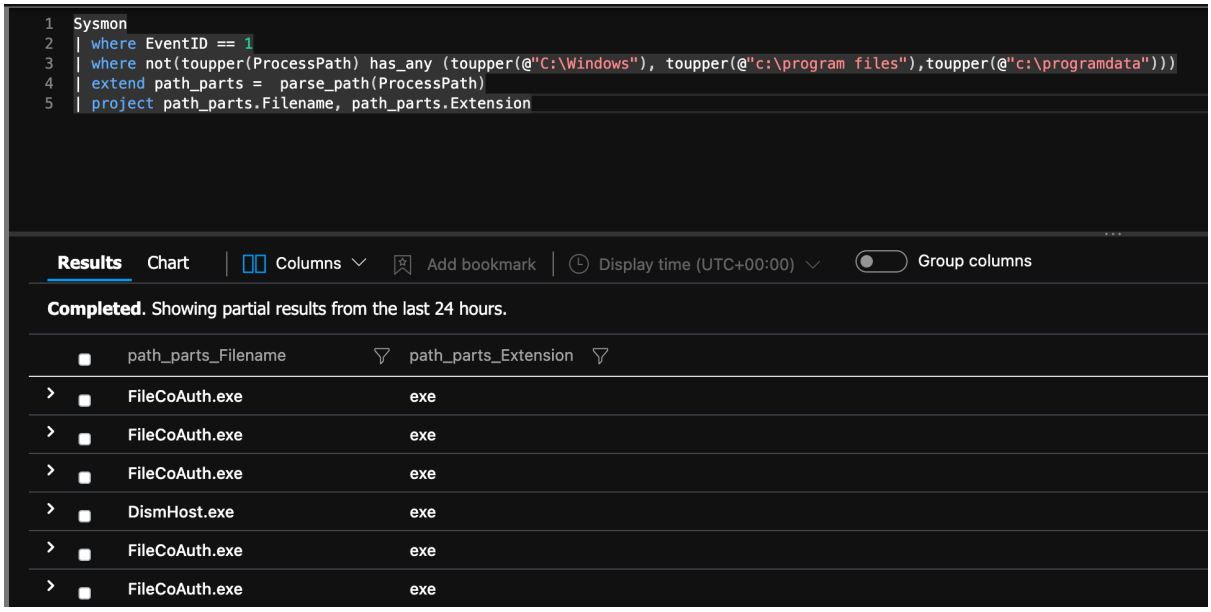
The parse_path function can be used to parse a file path and return a dynamic object (array) containing the following parts of the path:

- Scheme
- RootPath
- DirectoryPath
- DirectoryName
- FileName
- Extension
- AlternateDataStreamName

```
Sysmon
| where EventID == 1
| where not(toupper(ProcessPath) has_any (toupper(@"C:\Windows"), toupper(@"c:\program
files"),toupper(@"c:\programdata")))
| extend path_parts =  parse_path(ProcessPath)
| project path_parts
```



We can then use the parsed elements to select which one we need.

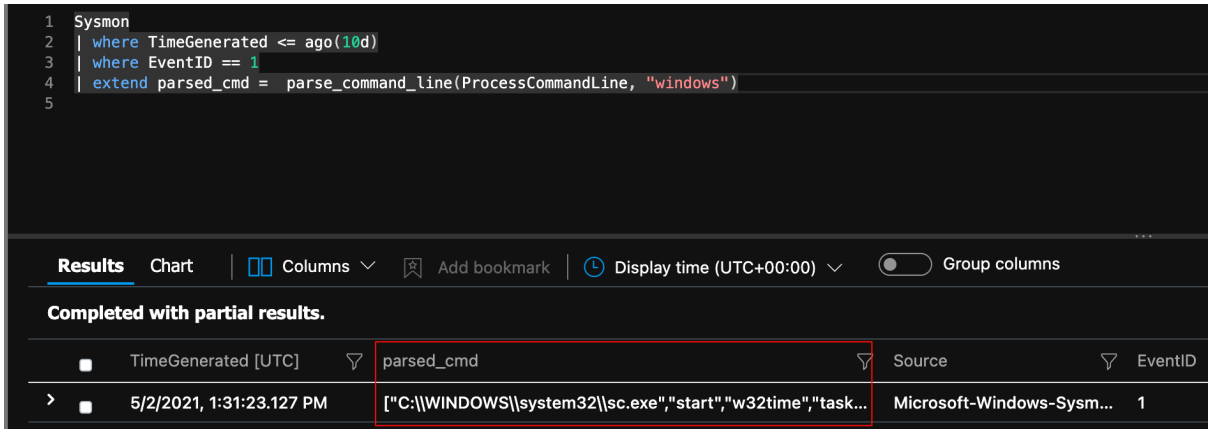


Parse_command_line

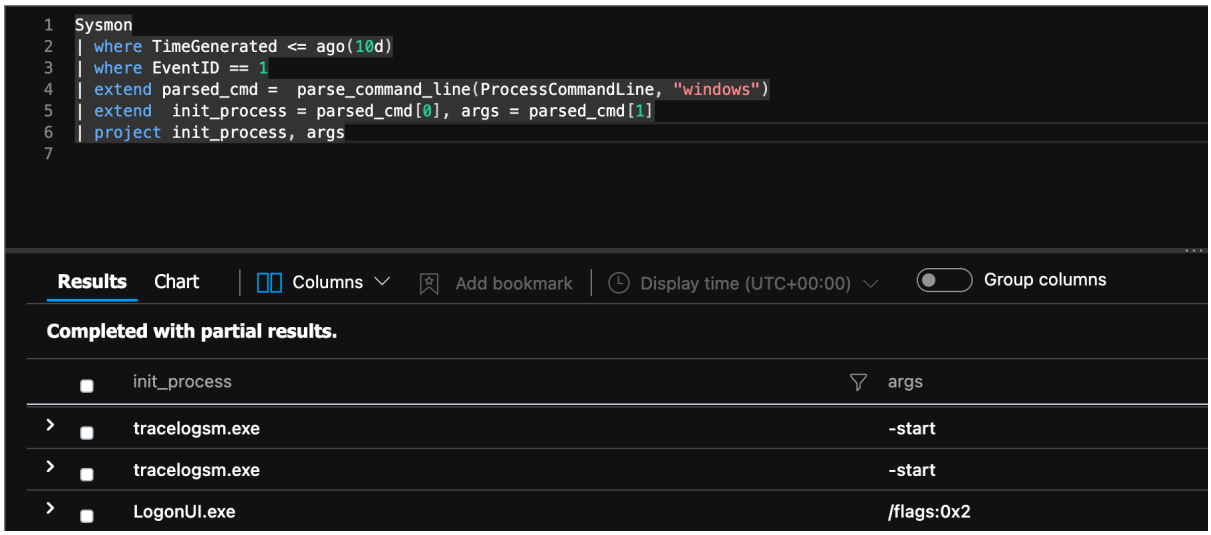
Similar to the *parse_path* function the *parse_command_line* parses a command line input and returns an array of the command line elements.

> ⚠ The only available parser is "windows"

```
Sysmon
| where TimeGenerated <= ago(10d)
| where EventID == 1
| extend parsed_cmd =  parse_command_line(ProcessCommandLine, "windows")
```



Since this is an array of elements we can select which ones to output.



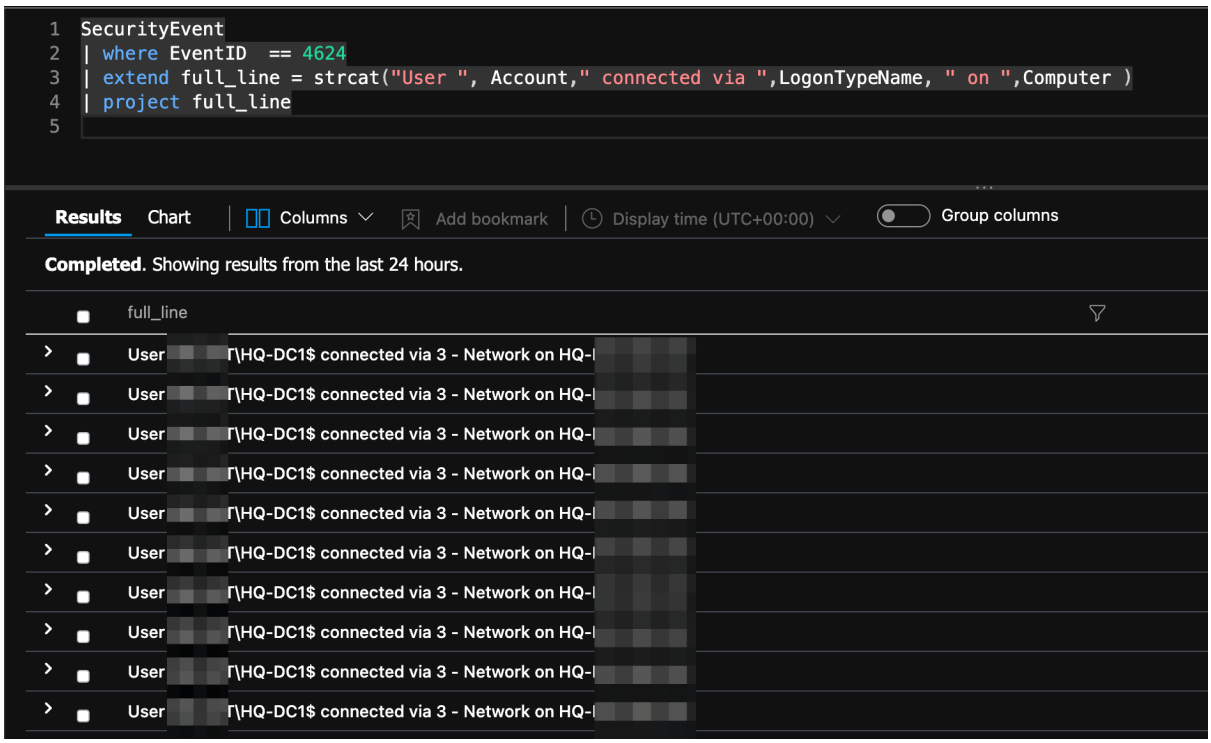
Strcat

Strcat can be used to concatenate strings.

```
print str = strcat("hello", " ", "world")
```

We can perform more complex string concatenations as below

```
SecurityEvent
| where EventID  == 4624
| extend full_line = strcat("User ", Account," connected via ", LogonTypeName, " on ",
Computer )
| project full_line
```



Set_has_element

The *set_has_element* operator determines if a set contains a specified element and returns true or false.

```
let lolbins = dynamic(["advpack.dll","at.exe","atbroker.exe","bash.exe","bitsadmin.ex
e","Comsvcs.dll","devtoolslauncher.exe","diantz.exe","diskshadow.exe","dllhost.exe","D
nscmd.exe","dxcap.exe","Esentutl.exe","eventvwr.exe","expand.exe","explorer.exe","extr
ac32.exe","findstr.exe","forfiles.exe","ftp.exe","Gpscript.exe","hh.exe","Ie4uinit.ex
e","Ieadvpack.dll","ieframe.dll","Infdefaultinstall.exe","makecab.exe","manage-bde.ex
e","manage-bde.wsf","mavinject.exe","mmc.exe","msconfig.exe","msdt.exe","mshta.exe","M
shtml.dll","msiexec.exe","netsh.exe","ntdsutil.exe","odbcconf.exe","pcalua.exe","pcwru
n.exe","Pcwutl.dll","pktmon.exe","pnputil.exe","Presentationhost.exe","presentationhos
t.exe","print.exe","psr.exe","Rasautou.exe","reg.exe","regedit.exe","regini.exe","Regi
ster-cimprovider.exe","regsvcs.exe","regsvr32.exe","replace.exe","rpcping.exe","rundll
32.exe","runonce.exe","sc.exe","schtasks.exe","scriptrunner.exe","setupapi.dll","Setup
api.dll","Shdocvw.dll","shell32.dll","SyncAppvPublishingServer.exe","SyncAppvPublishin
gServer.vbs","Syssetup.dll","Ttdinject.exe","tttracer.exe","url.dll","verclsid.exe","v
sjitdebugger.exe","winrm.vbs","wscript.exe","wsl.exe","wsreset.exe","wuauclt.exe","xwi
zard.exe","Zipfldr.dll"]);
Sysmon
| where EventID == 1
| where set_has_element(lolbins,FileName) == True
| project FileName
```

The example above checks if the *lolbins* set contains the *Filename* from Event ID 1. This will individually check each entry in the lolbins set against the *FileName* column from Sysmon Event ID 1 and return only the results where the condition is True.

Aggregation Functions

Any

The *any* function can be used for randomly selecting a record from the results.

```
SecurityEvent
| where TimeGenerated between (startofday(datetime(2021-03-01))..endofday(datetime(202
1-04-10)))
| where EventID == 4663
| summarize any(ObjectName)
```

Multiple columns can also be included

```
SecurityEvent
| where TimeGenerated between (startofday(datetime(2021-03-01))..endofday(datetime(202
1-04-10)))
| where EventID == 4663
| summarize any(ObjectName)
```

Or a wildcard be used to randomly select from all columns

```
SecurityEvent
| where TimeGenerated between (startofday(datetime(2021-03-01))..endofday(datetime(202
1-04-10)))
| where EventID == 4663
| summarize any(*)
```

Dcount

Returns an estimate for the number of distinct values that are taken by a scalar expression in the summary group. It is important to note that *dcount* trades performance of accuracy therefore different results may be returned after each execution.

```
SecurityEvent
| where TimeGenerated between (startofday(datetime(2021-03-01))..endofday(datetime(202
1-04-10)))
| where EventID == 4624
| summarize event_per_user = dcount(Computer) by Account
```

The above query will perform a distinct count using the *Computer* column and summarise the results based on the Account column.

The *dcount* operator can be combined with other operators as well such as *bin* to return the distinct bin values.

```
SecurityEvent
| where TimeGenerated between (startofday(datetime(2021-03-01))..endofday(datetime(202
1-04-10)))
| where EventID == 4624
| summarize event_per_user_time = dcount(bin(TimeGenerated,1h)) by Account
```

Countif

The *countif* operator performs a count of the results for which the statement inside the parenthesis returns True.

An example using the the *countif* operator would be to perform a count on Sysmon Event ID 1 only where the *Filename* contains the string "cscript" and summarize the results by *DeviceName*.

```
Sysmon
| where EventID == 1
| summarize countif(FileName has "cscript") by DeviceName
```

In the example below we perform a count only if the *dst_ip* does not start with "192.168"

```
ZeekDNS
| where not(ipv4_is_match(dst_ip,"8.8.8.8")) and not(ipv4_is_match(dst_ip,"10.10.10.1/
24"))
| summarize external_ips = countif(not(dst_ip startswith "192.168")) by src_ip
```

Further more complex combinations can be made in the query.

```
Sysmon
| where EventID == 1
| summarize countif(FileName !in ((Sysmon | where EventID  == 5 | project ProcessPat
h)))
```

If the example above the query will return only the results where the *FileName* column from Sysmon Event ID 1 does not exist in the *ProcessPath* of Sysmon Event ID 5. Essentially this query will return the results where a process has been started but has not been terminated. This example was shown to explain how a subquery can be used inside a query.

Dcountif

The *dcountif* operator returns the number of distinct values if the expression results to true.

```
Sysmon
| where EventID == 1
| summarize distinc_count = dcountif(DeviceName,FileName == "cscript.exe") by DeviceNa
me
```

The query above will return the distinct count of the *DeviceName* column when the FileName is equal to "cscript.exe" returns true.

Make_list

The make_set operator returns a json dynamic array of all values.

⚠ Can only be used inside the *summarize* operator

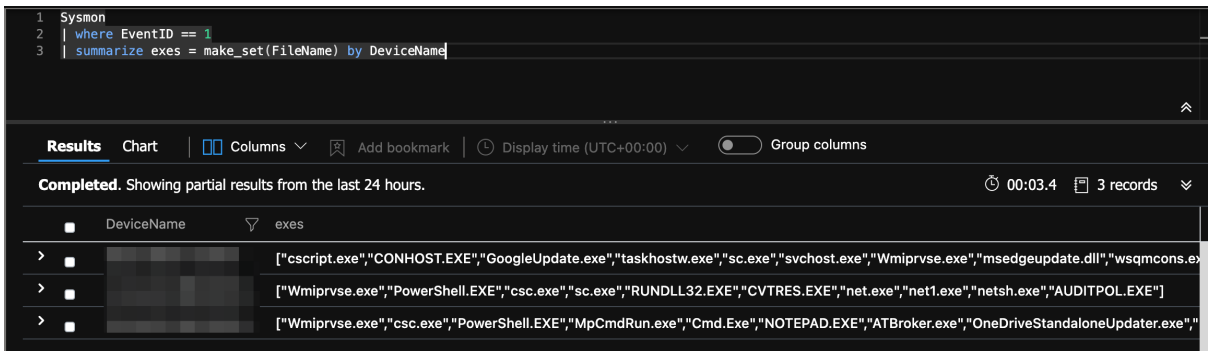
```
Sysmon
| where EventID == 1
| summarize exes = make_list(FileName) by DeviceName
```

Make_set

Similar to the *make_list* operator the *make_set* operator returns a json dynamic array of distinct values.

⚠ Can only be used inside the *summarize* operator

```
Sysmon
| where EventID == 1
| summarize exes = make_set(FileName) by DeviceName
```



User analytics

Active_users_count

The *active_users_count* plugin calculates a distinct count of values where each value appears at least a number of times i a specific period.

The following query will perform a users count using Event ID 4624, i.e users that performed a log in and appeared in the logs at least 3 times in a window of 30 days.

```
let start = datetime(2021-04-01);
let end = datetime(2021-05-20);
let bin = 7d;
let loopbackwindow = 30d;
let period = 1d;
let ActivePeriods = 3;
SecurityEvent
| where EventID == 4624
| evaluate active_users_count(Account, TimeGenerated, start, end,loopbackwindow, perio
d, ActivePeriods,bin)
```

A user is considered active if it fulfills both of the following criteria:

- The user was seen in at least three distinct days (Period = 1d, ActivePeriods=3).
- The user was seen in a lookback window of 8 day before and including their current appearance.

Sliding_Window_Counts

The *sliding_window_counts* plugin calculates counts and distinct count of values in a sliding window over a lookback period.

In the following example we want to count of users that logged in in a period of time but also want to see how many unique users performed those logins.

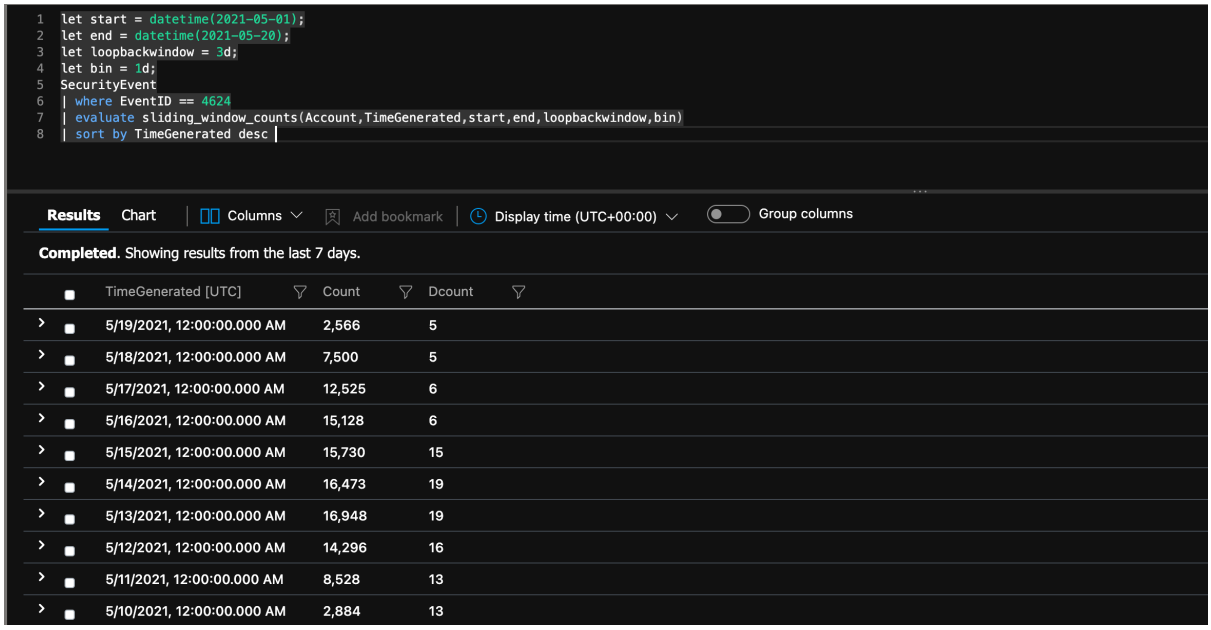
The *sliding_window_counts* plugin is very useful to perform this type of analysis.

```
let start = datetime(2021-05-01);
let end = datetime(2021-05-20);
let loopbackwindow = 3d;
let bin = 1d;
SecurityEvent
| where EventID == 4624
| evaluate sliding_window_counts(Account,TimeGenerated,start,end,loopbackwindow,bin)
| sort by TimeGenerated desc
```

As a first step the timeperiod is defined through the start and end variables. The loopback window is defined so that the query will use the start date and go back accordingly to how many days/hours/seconds are defined in the *loopback* window.

The *sliding_window_counts* takes as input:

- The column to perform the count and dcount on
- The timeline column
- The start and end variable names
- The loopback period
- The aggregation method, in this case daily bins



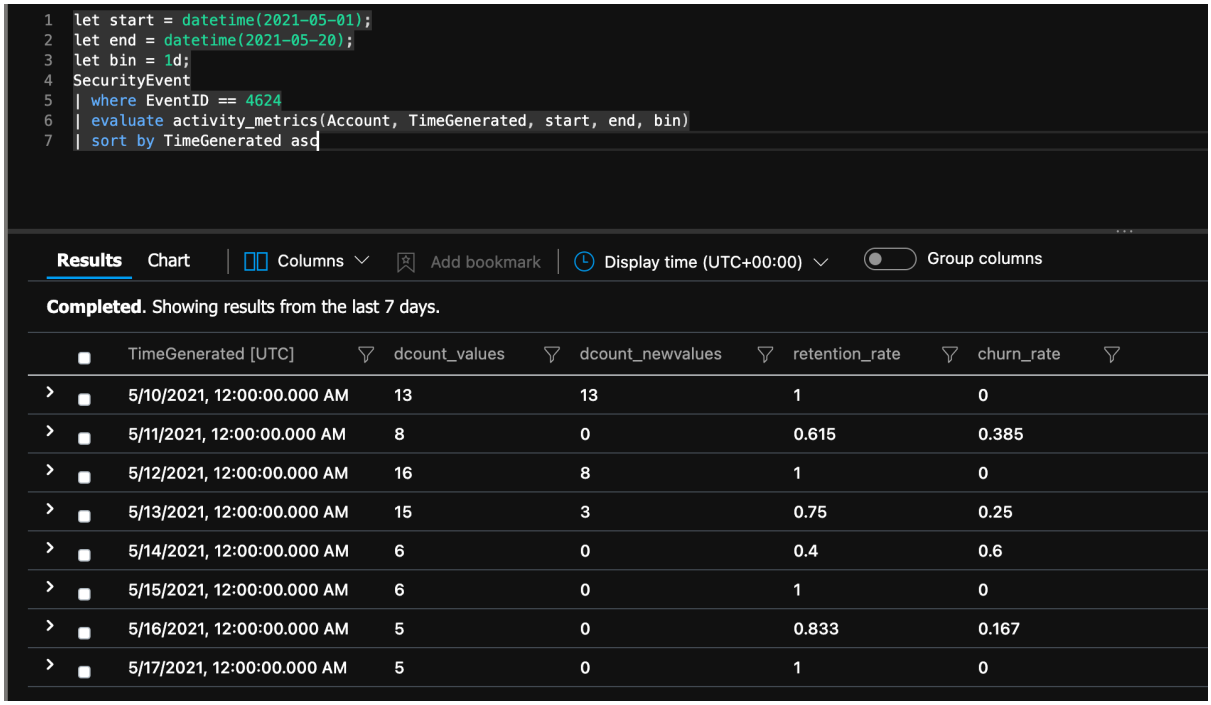
The output of this query shows the number of counts of 4624 events and distinct counts based on the Account column. Taking the first row as example, on that specific date there were 2566 events originating from 5 distinct Accounts.

Activity_metrics

The *activity_metrics* plugin calculates metrics comparing the current and previous time periods.

```
let start = datetime(2021-05-01);
let end = datetime(2021-05-20);
let bin = 1d;
SecurityEvent
| where EventID == 4624
| evaluate activity_counts_metrics(Account, TimeGenerated, start, end, bin)
| sort by TimeGenerated asc
```

Since the defined period is 1d that means the metrics will be performed on a daily basis.



- The dcount values is the dinstinc count of the current period
- Dcount_newvalues refers to the next period distinct counts
- Retention rate is the total number of items divided by the items in the beginning of the period
- Retention rate 0.0 to 1.0
    - The highest score refers a large number of entries contained in the current period and also where present in the previous period.
- Churn rate
    - Can vary from 0.0 to 1.0. A larger number refers to entries being present in the previous period but not in the next (lost)

Activity_counts_metrics

The activity_counts_metrics calculates activity metrics for each time window compared/aggregated to all previous time windows

```
let start = datetime(2021-05-01);
let end = datetime(2021-05-20);
let bin = 1d;
```

```
SecurityEvent
| where EventID == 4624
| evaluate activity_counts_metrics(Account, TimeGenerated, start, end, bin)
| sort by TimeGenerated asc
```