KQL - Advanced

This guide shows you how to use datatypes, functions and more.

Content

Datatypes

datetime

Description: The datetime (date) data type represents an instant in time, typically expressed as a date and time of day. Every table has a column *TimeGenerated* to know when a record is created. With it you could sort, filter and format the entries.

To **sort** a table by a datetime column use the operator *sort/order*. Example:

```
T | sort by TimeGenerated asc
```

To **filter** a table by datetime use the *where* operator and the *format_datetime* function. Keep in mind, that datetime always consist of a date *and* a time of day, even if you are using datetime(). Example:

```
T | where TimeGenerated == datetime(anyvalidvalue)

Heartbeat
| where TimeGenerated <= now() // this is important to set the time range of the
query to 'Set in query'.
| where (format_datetime( TimeGenerated, 'yyyy-M-d')) ==
format_datetime(datetime(2022-5-12), 'yyyy-M-d')
// The result of the query is the table of all heartbeats from May, 12th 2022 to
any time on that day.
```

To extract the date of a datetime value use the `startofday()` function.

```
let yesterday = startofday(now(-1d));
print yesterday
```

> For additional format specifier see the documentation of format_datetime().

To **format** dates and times use the operator *project* and the function *format_datetime()*. Example:

```
T | project format_datetime(TimeGenerated, 'dd-MM-yy')

T | project Date = format_datetime(TimeGenerated, 'dd-MM-yy')
```

The following tables gives you an overview of other datetime functions:

Function	Purpose	Syntax	Notes
datetime_add()	Adds periods to a datetime	datetime_add(period, amount, datetime)	*Supported periods*: year, quarter, month, week, day, hour, minute, second, millisecond, microsecond, nanosecond

Function	Purpose	Syntax	Notes
datetime_diff()	Calculates the number of the specified periods between two datetime values. The result type is an integer, representing the amount of periods.	datetime_diff(period, datetime_1,datetime_2)	*Supported* `periods`: year, quarter, month, week, day. hour, minute, second, millisecond, microsecond, nanosecond.
datetime_part()	Extracts the requested date part as an integer value.	datetime_part(part,datetime)	*Supported* `parts`: Year, Quarter, Month, week_of_year, Day, DayOfYear, Hour, Minute, Second, Millisecond, Microsecond, Nanosecond.
dayofmonth()	Returns the integer number representing the day number of the given month	dayofmonth(datetime())	
dayofweek()	Returns the integer number of days since the preceding Sunday	dayofweek(datetime)	
dayofyear()	Returns the integer number represents the day number of the given year.	dayofyear(datetime)	

Example:

```
print datetime_part("Day", datetime(2022-5-12))

print dayofweek(datetime(2022-5-12))
//returns 4 indicating Thursday.
```

To create variable of type datetime with custom values use the function make_datetime().

```
let xmas = make_datetime(2023,12,24);
print xmas
```

Typical queries with datetime conditions:

1. Which data records were created yesterday?
    - Use the function now() and startofday() to create a variable with yesterdays date: `startofday(now(-1d))`
2. Which data records were created yesterday between 1pm and 2pm?
    - `datetime_add("hour", 1, startofday(now(-1d)))`
3. Which data records were created on a specific day between 8am and 9am?
    - `datetime(2022-5-12 8am)`

timespan

The timepan(time) data type represents a time interval. Some examples are 2d, 3h, 5m 10s. More see here.

A timespan could be used for calculations and in the functions ago() or now().

Table operators

union

join

evaluate

The evaluate operator is a tabular operator that provides the ability to invoke query language extensions known as plugins.

```
T | evaluate PluginName
```

The following example shows how to unpack a dynamic column:

```
datatable (d:dynamic)
[
    dynamic({"Name": "Mercury", "OS":"Windows Server", "RAM": 4}),
    dynamic({"Name": "Venus", "OS":"Windows Server", "RAM": 8}),
    dynamic({"Name": "Earth", "OS":"Linux", "RAM": 4}),

]
| evaluate bag_unpack(d)
```

More information about bag_unpack() could be found in the documentation.

parse

Importing external data

The tabular operator `externaldata()` allows you to work with data saved not in the workspace. It returns a table whose schema is defined in the query itself, and whose data is read from an external storage artifact, such as a blob in Azure Blob Storage or a file in Azure Data Lake Storage.

Syntax:

```
externaldata ( ColumnName : ColumnType [, ...] )
[ StorageConnectionString [, ...] ]
[with ( PropertyName = PropertyValue [, ...] )]
```

The file computer.csv is stored as a blob in a storage account and has the following content:

Name	OS	RAM	IP	CPU
Merkury	Windows 11	8	10.20.0.4	4
Venus	Windows Server 2022	16	10.10.0.12	8
Mars	Windows Server 2022	8	10.10.0.13	8
Jupiter	Linux	48	10.10.0.14	24

To get all data from the file use:

```
let myExternalComputers = externaldata (Name:string, OS:string, RAM:int,
IP:string, CPU:int)
[@"https://distsacoursestaff.blob.core.windows.net/demodata/computers.csv"]
with (ignoreFirstRecord=true);
myExternalComputers
```

To get only some columns use:

```
let myExternalComputers = externaldata (Name:string, RAM:int, CPU:int)
[@"https://distsacoursestaff.blob.core.windows.net/demodata/computers.csv"]
with (ignoreFirstRecord=true);
myExternalComputers
```

> Hint: The order of column definition does not matter

The next example gets a list of know IPv4 address related to their location:

```
let IP_Data =
external_data(network:string,geoname_id:long,continent_code:string,continent_name:
string
,country_iso_code:string,country_name:string,is_anonymous_proxy:bool,is_satellite_
provider:bool)
```

```
    ['https://raw.githubusercontent.com/datasets/geoip2-ipv4/master/data/geoip2-
ipv4.csv'];
```

count

The operator `count` returns the numbe of records in the input record set. It should be the latest in the data flow model.

```
DeviceInfo | count
```

getschema

The operator `getschema` produces a table that represents a tabluar schema of the input. It should be the latest in the data flow model.

```
DeviceInfo | getschema
```

Scalar Functions

String Functions

Name	Example	Description
tostring()	tostring(345) == "345"	Converts input to a string representation.
string_size()	`string_size(column|"value")`	Returns the size of a string.
strcat()	`print str = strcat("hello", " ", "world")`	Concatenates between 1 and 64 arguments. If the arguments aren't of string type, they'll be forcibly converted to string.
tolower()	`tolower("Hello World!")` == "hello world!"	~
toupper()	`toupper("Hello World!")` == "HELLO WORLD!"	~
trim(regex, source)	`trim("_", "_Hello World!_")`; `trim("_+", "__Hello World!__")`	Removes all leading and trailing matches of the specified regular expression.
trimstart(regex, source)	~	Removes leading match of the specified regular expression.
trimend(regex, source)	~	Removes trailing match of the specified regular expression.

Name	Example	Description
split(source, delimiter[, requestedIndex])	`split("The quick brown fox."," ")` == ["The","quick","brown","fox."]	Splits a given string according to a given delimiter and returns a string array with the contained substrings. This function is case-sensitive.
substring(source, startingIndex[, length])	`substring("Hello World!", 0, 5)` == "Hello"	Extracts a substring from a source string starting from some index to the end of the string.
replace_string(source, searchtext, replacetext)	`replace_string("Hello World!", "World", "Universe")`	Replaces all string matches with another string. This function is case-sensitive.
translate(searchlist, replacementlist, source)	`translate("eo","x","Hello World!")` == "Hxllx Wxrld!"; `translate("krasp", "otsku", "spark")` == "kusto"	Replaces a set of characters ('searchList') with another set of characters ('replacementList') in a given a string. The function searches for characters in the 'searchList' and replaces them with the corresponding characters in 'replacementList'. This function is case-sensitive.
extract()	~	Get a match for a regular expression from a source string. See this section..

extract()

If you have to extract any text of a column you could use the function `extract()`. To build the regex let you help with that site: https://regex101.com.

This function has three mandatory parameters:

Parameter	Type	Description
regex	string	A regular expression.
captureGroup	int	The capture group to extract. 0 stands for the entire match, 1 for the value matched by the first '('parenthesis')' in the regular expression, and 2 or more for subsequent parentheses.
source	string	The string to search.

Example 1: This example shows you how to extract the hostname of a FQDN:

```
VMComputer
| extend Computername = extract(@"(^[a-z\|A-Z\|0-9]+)", 0, Computer)
| project Computer, Computername
```

Example 2: In the next example you will see how to extract the domain name also.

```
VMComputer
| extend Computername = extract(@"(^[a-z\|A-Z\|0-9]+)", 0, Computer),
         Domainname = extract(@"(^[a-z\|A-Z\|0-9]+)\.(.+)$",2,Computer)
| project Computer, Computername, Domainname
```

Aggregation Functions

make_list() vs. make_set()

This function creates an array of column entries.

```
datatable (city:string) ["Jupiter","Mercury","Venus","Mercury"]
| summarize cities = make_list(city)
```

After you executed the above query you will see that the result array has as many entries as the table has records. In other words, it could be to get entries doubled in the array.

Use the `make_set()` function to avoid doublets:

```
datatable (city:string) ["Jupiter","Mercury","Venus","Mercury"]
| summarize cities = make_set(city)
```

Techniques

Working with arrays

Create an array

To create an array you could use the dynamic() function or aggregation functions like `make_list()` or `make_set()`.

Example:

```
let planets = dynamic(["Jupiter","Mercury","Venus"]);
print planets
// the result is a scalar array.

datatable (planets:string) ["Jupiter","Mercury","Venus"] | summarize Planets =
make_list(planet)
// the result is a table with a single column 'Planets'.
```

Both results of the above examples could be used with `where` and the `in ()` operator:

```
T
| where AnyColumn in (planets)
```

Get values of an array

To reference a value of an array use the following syntax:

```
arrayname[index]
```

Additional array functions

function	Syntax	Purpose
array_sort_asc()	`array_sort_asc(array1,array2,...)`	Receives one or more arrays. Sorts the first array in ascending order. Orders the remaining arrays to match the reordered first array.
array_slice()	`array_slice(arr, startindex, endindex)`	Extracts a slice of a dynamic array.
array_shift_left(), array_shift_right()	`array_shift_left(arr, shift_count [, defaultValue])`	Shifts array to the left/right, kicks out starting/ending values by adding the defaultValue for the free positions.
array_rotate_left(), array_rotate_right()	`array_rotate_left(arr, rotatecount)`	Rotates values inside a dynamic array to the left/rigth.

Example:

```
let planets = dynamic(["Mercury", "Venus", "Earth", "Mars", "Jupiter", "Saturn",
"Uranos","Neptune"]);
range x from 0 to 7 step 1
| extend Planet = planets[x]

let planets = dynamic(["Mercury", "Venus", "Earth", "Mars", "Jupiter", "Saturn",
"Uranos","Neptune"]);
let planetsrotated = array_rotate_left(planets,1);
range x from 0 to (array_length(planets) - 1) step 1
| extend Planet = planets[x],
         PlanetsRotated = planetsrotated[x]
```

Arrays saved in columns of a table

If an array is saved in a column use the `mv-expand` Operator to get each array entry in a single line for further analysis.

```
let myHosts = datatable (Hostname:string, Memory:real, CPU:int, Type:string,
Features:dynamic ) [
    "Jupiter",16384,32,"Server", dynamic(["ExSrv","DC","DNS"]),
    "Mars",8192,8,"Server", dynamic(["SPS","DHCP"]),
    "Mercury",4096,4,"Server",dynamic(["DC","DNS","DHCP"]),
    "Venus", 8192,8,"Server",dynamic(["FS","PS","Backup"]),
    "Saturn", 8192,8,"Server",dynamic(["FS","Backup","DC","DNS"]),
    "PC01",4096,4,"Client", dynamic(null) ];
myHosts
| where isnotempty(Features)
| mv-expand Features
// To get the amount of instances of each feature add the next to lines.
// The first line is required to change the datatype to string since summarize
cannot work
// with a dynamic column.
| extend tostring(Features)
| summarize count() by Features
```

Working with JSON objects / property bags

Create or get a JSON object / property bag

To create a json object use the dynamic() function. Similar to creating an array you have to define you json object enclosed in curley brackets `{ }`. Use `"attributename":value` to define the attribute and seperate multiple with a comma `"attributename1":value, "attributename2":value`.

```
let myHostJSON =
dynamic({"Hostname":"Jupiter","RAM":4096,"CPU":8,"OSType":"Windows Server"});
print Host=myHostJSON
```

To create multiple json objects, use the same function to create an array of json objects:

```
let myServersJSON =
dynamic([{"Hostname":"Jupiter","RAM":4096,"CPU":8,"OSType":"Windows Server"},
{"Hostname":"Saturn","RAM":2048,"CPU":4,"OSType":"Linux"}]);
print Host=myServersJSON
```

Working with JSON objects / property bag

Often a property bag is saved in a source table in a column with the type of dynamic. To expand a property bag into a table you could use `mv-expand`, `project` or `bag_unpack()`:

The operator `mv-expand` converts an array into multiple lines or converts a property bag into multiple lines (one line for each property).

Example:

```
AzureActivity
| project Authorization_d
| mv-expand Authorization_d
```

> Note: The result is a table of all properties of the bag and each of them is record set. In other words: if your bag would have 5 attributes, mv-expand would produce 5 record sets.

Example:

```
AzureActivity
| project Caller, Authorization_d
| mv-expand Authorization_d
| project Caller, Authorization_d
```

> Note: The result shows now also the content of the column *Caller* as many times as you would have properties per bag.

The plug-in bag_unpack() creates a table. One column for each property of the bag.

Example:

```
AzureActivity
| project Caller, Authorization_d
| evaluate bag_unpack(Authorization_d)
```

> Note: The result is a conversion. The property bag was converted to colums which are added to table. This allows you to use operators like `project`, `where` or `extend` in a handy way.

This datatable consists of a column *Hardware* with a single json object and a column *Software* with an array of multiple json objects.

```
let myHosts = datatable (Hostname:string, Hardware:dynamic , Software:dynamic ,
Location:string ) [
    "Jupiter",
dynamic({"RAM":8192,"CPU":12,"DiskCapacityGB":512,"DiskCount":6,"DiskVendor":"WD"}
),
dynamic([{"Product":"Exchange","Vendor":"Microsoft","Version":"2019","BuiltIn":fal
se},
{"Product":"FileServer","Vendor":"Microsoft","Version":"2022","BuiltIn":true}]),
"Graz",
    "Saturn",
```

```
dynamic({"RAM":8192,"CPU":8,"DiskCapacityGB":768,"DiskCount":12,"DiskVendor":"WD"}
),
dynamic([{"Product":"SharePoint","Vendor":"Microsoft","Version":"2019","BuiltIn":f
alse},{"Product":"DC","Vendor":"Microsoft","Version":"2022","BuiltIn":true},
{"Product":"DNS","Vendor":"Microsoft","Version":"2022","BuiltIn":true}]), "Graz",
    "Neptune",
dynamic({"RAM":4096,"CPU":2,"DiskCapacityGB":256,"DiskCount":4,"DiskVendor":"IBM"}
), dynamic([{"Product":"S/4HANA","Vendor":"SAP","Version":"2021","BuiltIn":false},
{"Product":"PrintServer","Vendor":"Microsoft","Version":"2022","BuiltIn":true}]),
"Linz",
    "Uranos",
dynamic({"RAM":6144,"CPU":8,"DiskCapacityGB":512,"DiskCount":8,"DiskVendor":"IBM"}
),
dynamic([{"Product":"Exchange","Vendor":"Microsoft","Version":"2019","BuiltIn":fal
se},{"Product":"DC","Vendor":"Microsoft","Version":"2022","BuiltIn":true},
{"Product":"DNS","Vendor":"Microsoft","Version":"2022","BuiltIn":true}]), "Linz",
];
myHosts
```

Hostname	Hardware	Software	Location
Jupiter	{"RAM":8192,"CPU":12,"DiskCapacityGB":512,"DiskCount":6,"DiskVendor":"W...	[{"Product":"Exchange","Vendor":"Micros...	Graz
Hostname	Jupiter		
Hardware	{"RAM":8192,"CPU":12,"DiskCapacityGB":512,"DiskCount":6,"DiskVendor":"WD"}		
CPU	12		
DiskCapacityGB	512		
DiskCount	6		
DiskVendor	WD		
RAM	8192		
Software	[{"Product":"Exchange","Vendor":"Microsoft","Version":"2019","BuiltIn":false},{"Product":"FileServer","Vendor":"Microsoft","Version":"2022","BuiltIn":...		
0	{"Product":"Exchange","Vendor":"Microsoft","Version":"2019","BuiltIn":false}		
1	{"Product":"FileServer","Vendor":"Microsoft","Version":"2022","BuiltIn":true}		
Location	Graz		
Saturn	{"RAM":8192,"CPU":8,"DiskCapacityGB":768,"DiskCount":12,"DiskVendor":"WD"}	[{"Product":"SharePoint","Vendor":"Micros...	Graz
Neptune	{"RAM":4096,"CPU":2,"DiskCapacityGB":256,"DiskCount":4,"DiskVendor":"IBM"}	[{"Product":"S/4HANA","Vendor":"SAP","Ve...	Linz
Uranos	{"RAM":6144,"CPU":8,"DiskCapacityGB":512,"DiskCount":8,"DiskVendor":"IBM"}	[{"Product":"Exchange","Vendor":"Microsof...	Linz

Should you find an array of property bags in a table, use the `mv-expand` operator in conjunction with the `bag_unpack()` function. In the next example the first `mv_epxand` operator expands the array of the bags, the second expands each property.

```
let myHosts = datatable ( ... ) ... ;
myHosts
| mv-expand Software
| evaluate bag_unpack(Software)
```

Would you like now to expand the *Hardware* columns too, use `bag_unpack()` twice:

```
let myHosts = datatable ( ... ) ... ;
myHosts
| mv-expand Software
| evaluate bag_unpack(Software)
| evaluate bag_unpack(Hardware)
```

An other way to work with json objects is to use the operators `extend` or `project`. Both are able to create columns:

```
let myHosts = datatable ( ... ) ... ;
myHosts
| extend DiskVendor = Hardware.DiskVendor, DiskCapacity = Hardware.DiskCpacityGB
```

> Note:
>
> - `extend` adds a column to the table whereby `project` creates new columns and replaces all others.
> - This technique could not be used with arrays of json objects.
> - This technique could also be used with the operator `where`.

IPv4 lookup

It is possible to use a plug in to find information about network locations given an IP address of a host. The following example uses tow tables. The table *Location_IP* holds information about networks; here just the geographical location. But it could be extended by any other details. The second table, *Hosts*, holds information about used IP addresses. The query itself executes a plug in `ipv4_lookup` with the tables and corresponding columns: **IPAddess** is checked against **Network**.

```
let Location_IP = datatable (Network:string, City:string, Country:string)
[
   "10.10.0.0/16","Graz","Austria",
   "10.20.0.0/16","Linz","Austria",
   "10.30.1.0/24","Munic","Germany",
   "10.30.2.0/24","Hamburg","Germany",
   "10.30.3.0/26","Nice","France"
];
let Hosts = datatable(Hostname:string,IPAddress:string,OSType:string )
[
   "Jupiter","10.10.0.100","Windows Server",
   "Mercury","10.30.2.50","Linux",
   "Venus","10.20.0.254","Windows Server",
   "Mars","10.30.3.12","Windows Server",
   "Earth","10.30.3.100","Linux"
];
Hosts
| evaluate ipv4_lookup(Location_IP,IPAddress,Network, return_unmatched = true)
```

Working with multiple workspaces

Sometimes it could be your date are distributed in different workspaces. To reference a table in an other workspace use the function workspace():

```
workspace("aztrg2207LogAnalytics").Heartbeat
// This returns the tabel Heartbeat of workspace 'aztrg2207LogAnalytics').
```

> Hint: The parameter indicates the workspace. You could use workspacename, Azure Resource ID or workspace ID. For more information use this blog or the documentation.

As long as you are using the same schema you could use the `union` operator to create a single-in-memory table. Some examples:

```
let ws1 = workspace("aztrg2207LogAnalytics").Heartbeat | extend Workspace =
'aztrg2207LogAnalytics';
let ws2 = Heartbeat | extend Workspace = 'aztrg2207LogAnalyticsSecond';
union ws1,ws2
// oder: ws1 | union ws2

workspace("aztrg2207LogAnalytics").Heartbeat
| union Heartbeat
```

Funtion materialize()

Description: This functions creates a tabular result and saves it in cache. For further use this cached result is used instead of quering the source again.

Example:

```
let T = DeviceInfo
name | where field == 'value'
```

Link: https://docs.microsoft.com/en-us/azure/data-explorer/kusto/query/materializefunction

Operator parse

```
let myPlanets = datatable (Item:int, Statement:string) [
    1, "The planet Mercury has a circumference of 15330 km and a mass of 0.33 E+24
kg.",
    2, "The planet Venus has a circumference of 38023 km and a mass of 4.87 E+24
kg.",
    3, "The planet Earth has a circumference of 40075 km and a mass of 5.97 E+24
kg.",
    4, "The planet Mars has a circumference of 21344 km and a mass of 0.64 E+24
```

```
kg." ];
myPlanets
| parse Statement with * "planet " planet:string " has a circumference of "
circum:long " km and a mass of " mass:real " E" *
| project Item,planet,circum,mass
```

```
let myHosts = datatable (Hostname:string, Hardware:dynamic , Software:dynamic ,
Location:string , Devices:dynamic ) [
    "Jupiter",
dynamic({"RAM":8192,"CPU":12,"DiskCapacityGB":512,"DiskCount":6,"DiskVendor":"WD"}
),
dynamic([{"Product":"Exchange","Vendor":"Microsoft","Version":"2019","BuiltIn":fal
se},
{"Product":"FileServer","Vendor":"Microsoft","Version":"2022","BuiltIn":true}]),
"Graz", dynamic(["Mouse","FIDO2-USB"]),
    "Saturn",
dynamic({"RAM":8192,"CPU":8,"DiskCapacityGB":768,"DiskCount":12,"DiskVendor":"WD"}
),
dynamic([{"Product":"SharePoint","Vendor":"Microsoft","Version":"2019","BuiltIn":f
alse},{"Product":"DC","Vendor":"Microsoft","Version":"2022","BuiltIn":true},
{"Product":"DNS","Vendor":"Microsoft","Version":"2022","BuiltIn":true}]), "Graz",
dynamic(["Mouse","iPhone"]),
    "Neptune",
dynamic({"RAM":4096,"CPU":2,"DiskCapacityGB":256,"DiskCount":4,"DiskVendor":"IBM"}
), dynamic([{"Product":"S/4HANA","Vendor":"SAP","Version":"2021","BuiltIn":false},
{"Product":"PrintServer","Vendor":"Microsoft","Version":"2022","BuiltIn":true}]),
"Linz", dynamic(["Pen","iPhone"]),
    "Uranos",
dynamic({"RAM":6144,"CPU":8,"DiskCapacityGB":512,"DiskCount":8,"DiskVendor":"IBM"}
),
dynamic([{"Product":"Exchange","Vendor":"Microsoft","Version":"2019","BuiltIn":fal
se},{"Product":"DC","Vendor":"Microsoft","Version":"2022","BuiltIn":true},
{"Product":"DNS","Vendor":"Microsoft","Version":"2022","BuiltIn":true}]), "Linz",
dynamic(["Mouse","FIDO2-USB"])
];
myHosts
| mv-expand Software
| evaluate bag_unpack(Software)
```