KQL Cheat Sheet v0.2.1 github.com/marcusbakker/KQL/blob/master/kql_cheat_sheet.pdf Bakk3rM

The purpose of this cheat sheet is to cover essential basics for the **K**usto **Q**uery **L**anguage (KQL). The majority of the queries from this cheat sheet will run on the `SecurityEvent` table: accessible via https://portal.loganalytics.io/demo. In the queries below, the table `SecurityEvent` is abbreviated by `T`. Many of the KQL functions and operators below link back the official KQL documentation.

The example queries only have the purpose of explaining KQL and may stop providing results due to changes in the data on the Log Analytics demo portal.

Generic

A string literal with a backslash requires **escaping** by a backslash: `"a string literal with a \\ needs to be escaped"`

The same can be achieved using a **verbatim string** literal by putting the @ sign in front: `@"a verbatim string literal with a \ that does not need to be escaped"`

More info on escaping string data types can be found here.

Add comments to your query with a double forward slash: `// This is a comment`

The `where` operator and the pipe (|) delimiter are essential in writing KQL queries.

`where` is used to filter rows from a table. In this example we filter on events from a source, the table `SecurityEvent`, where the column `Computer` is equal to "ContosoAppSrv1", and count the number of results: `SecurityEvent | where Computer == "ContosoAppSrv1" | count`

The pipe is used to separate data transformation operators. Such as: `where Computer == "ContosoAppSrv1"`. The result can be piped to a new operator. For example, to count the number for rows: `| count`

Only include **events from the last 24 hours** using the `ago()` function: `T | where TimeGenerated > ago(24h)`

For performance reasons always use time filters first in your query.

The `ago()` function supports multiple types of timespans. More info can be found here. For example:

`1d`	1 day
`10m`	10 minutes
`30s`	30 seconds

Include events that occurred **between a specific timeframe:** `T | where TimeGenerated between(datetime(2019-11-01 00:00:00) .. datetime(2019-11-01 06:00:00))`

Select and customize the columns from the resulting table of your query with the `project` operator.

- Specify the **columns to include**:
  `T | project TimeGenerated, EventID, Account, Computer, LogonType`
- **Rename columns**. In this example we renamed the column `Account` to `UserName`:
  `T | project TimeGenerated, EventID, UserName = Account, Computer, LogonType`
- **Remove columns** with `project-away`:
  `T | project-away EventSourceName, Task, Level`

Add calculated columns to the result using the `extend` operator: `T | extend EventAge=now()-TimeGenerated`

Count the number of records using the `count` operator: `T | count`

String search

Search across all tables and columns: `search "*KEYWORD*"`

- Keep in mind that this is a performance intensive operation.

Search for a specific value: `T | where ProcessName == @"C:\Windows\System32\regsvr32.exe"`

A **not equal to match** is done by adding an exclamation mark as prefix:

- Equal to: `==`
- Not equal to: `!=`

This is also supported in a similar way for other string operators.

A **case insensitive match** can be achieved using a tilde:

- Case sensitive: `==`
- Case insensitive: `=~`
- Case insensitive and not equal to: `!~`

This is also supported in a similar way for other string operators.

Match on values that **contain a specific string**: `T | where CommandLine contains "guest"`

Because `has` is more performant, it's advised to use `has` over `contains` when searching for full keywords. The following expression yields to `true`:

- `"North America" has "america"`

`contains` and `has` are case insensitive by default. A case sensitive match can be achieved by adding the suffix `_cs`: `contains_cs` / `has_cs`

Match on values **starting with or ending with a specific string**: `T | where Computer startswith "contoso"`

- Ending with a specific string: `endswith`

`startswith` and `endswith` are case insensitive by default. A case sensitive match can be achieved by adding the suffix `_cs`: `startswith_cs` / `endswith_cs`

Match on **multiple string values**: `T | where Computer in ("ContosoAppSrv1", "ContosoSQLSrv1")`

- Not equal to: `!in`
- Case insensitive: `in~`
- Case insensitive and not equal to: `!in~`

Match based on a **regular expression**: `T | where Computer matches regex "^Contoso.*"`

- KQL uses the re2 library and also complies with that syntax. Troubleshooting your regex can be done on regex101.com. Select the regex Flavor "Golang" which also makes use of re2.

A **not equal to match** can be done using the `not()` function: `T | where not(Computer matches regex "^Contoso.*")`

A **case insensitive match** can be achieved by providing the `i` flag: `T | where Computer matches regex "(?i)^contoso.*"`

Generic

Match based on conditions using logical operators. For example:

- `T | where EventID == 4624 and LogonType == 3`
- `T | where EventID == 4624 or EventID == 4625`
- `T | where (EventID == 4624 and LogonType == 3) or EventID == 4625`

Aggregate results from your query with the `summarize` operator:

- Aggregate on multiple columns:
  `T | summarize by Computer, Account`
- Aggregate on multiple columns and return the count of the group: `T | summarize count() by Computer, Account`

Besides `count()` many more very useful aggregation functions exist. An overview can be found here.

Sort the rows of the result using the `sort` operator:

```
T | where EventID == 4624 | summarize count() by
AuthenticationPackageName | sort by count_
```

By default, rows are sorted in descending order. Sorting in ascending order is also possible:

- `sort by count_ asc`
- Descending order: `desc`

Concatenate values. The result will be a string data type:

```
T | project example=strcat(EventID, " - ", Channel)
```

A variable number of values can be passed through the `strcat` function. If values are not a string, they will be forcibly converted to a string.

Numerical search

Search for a specific value: `T | where EventID == 4688`

- Not equal to: `!=`

All of the numerical operators can be found here.

Search for a **value less or greater than**: `T | where EventID == 4688 | summarize count() by Process | where count_ < 5`

- Greater: `>`
- Less or Equal: `<=`
- Greater or Equal: `>=`

Match on **multiple numeric values**:

```
T | where EventID in (4624, 4625)
```

Extract values

Extract values from a string or JSON data. For example, extract the “process name” using a regular expression (if you are less familiar with regular expressions have a look at the `split` and `parse` function):

```
SecurityAlert | extend _ProcessName=extract('"process
name": "(.*)"', 1, ExtendedProperties)
```

Because the column `ExtendedProperties` contains JSON data you can also use the function `extractjson()`:

```
SecurityAlert | extend _ProcessName =
extractjson("$.process name", ExtendedProperties)
```

If you need to extract multiple elements from JSON data, stored as a string, you can use the function `parse_json()`. Use the dot notation if the data is of the type `dictionary` or a list of `dictionaries` in an `array`. One way to find out is through the `gettype()` function. To play with data stored as a `dictionary` have a look at the help cluster in the Azure Data Explorer (table: `StormEvents`, column: `StormSummary`).

Named expressions and user-defined functions

Use the `let` statement to **bind names to expressions**. See below two examples of a named expression. Of course, much more complex expression can be created. Such as complete queries that can be nested inside another query (i.e. sub-query). For **sub-queries** consider the use of the `materialize()` function when the sub-query is called multiple times.

Take into account the semicolon at the end of the `let` statement:

-
  ```
  let _SearchWindow = ago(24h);
  T | where TimeGenerated > _SearchWindow
  ```

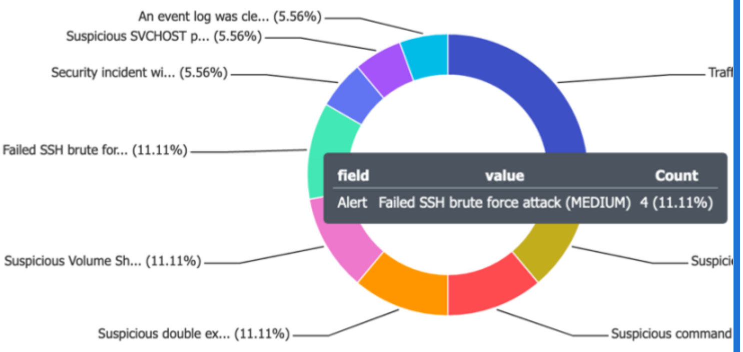
-
  ```
  let _computers = dynamic(["ContosoAppSrv1",
  "ContosoSQLSrv1"]);
  T | where Computer in (_computers)
  ```

The `let` statement can be used in many other useful ways. Such as to create **user-defined functions**. More info on the `let` statement can be found here.

Visualizations

The `render` operator can be used to **create visualizations**. Besides the below example, more types of visualizations are possible. More info can be found here. *(Pie charts are not the most telling graphics, but the support for the `render` operator is limited on the demo environment.)*

```
SecurityAlert |
summarize Count=count() by Alert=strcat(DisplayName,
" (", toupper(AlertSeverity), ")")
| sort by Count | render piechart
```



Join tables

KQL has the ability to **join tables**. In this example, we join some of the events in the `SecurityAlert` table with process creation events (event ID 4688) from the `SecurityEvent` table. More information on joining tables can be found here.

This query serves purely as an example to explain the `join` operator because all process data is contained within the column `Entities` of the `SecurityAlert` table.

```
SecurityAlert |
extend _ProcessId = extractjson("$.process id",
ExtendedProperties),
_ProcessCommandLine = tolower(extractjson("$.command
line", ExtendedProperties)),
_HostName = tolower(extractjson("$[0].HostName",
Entities))
| join kind=inner (
   SecurityEvent
   | where EventID == 4688
   | extend _HostName=tolower(Computer)
   | extend _ProcessCommandLine=tolower(CommandLine)
) on $left._ProcessId == $right.NewProcessId,
_HostName, _ProcessCommandLine
```