# Lesson 3 : Datasets

# Objectives

After completing this Learning, you will be able to:

**Understand different types of operators related to variables and datasets**

1. Datatable
2. Joins, Unions
3. Let, Set, Batch

# KQL Operators: Datatable

**KQL** has a number of built-in operators that allow you to interact with datasets and variables.

**The datatable operator** returns a table, whose values and schema you provide. Typically used with the **let** operator to keep a temporary table in memory.

**Usage:** datatable ( ColumnName : ColumnType [, ...] ) [ ScalarValue [, ScalarValue ...] ]

```
1  datatable (SomeInt:int, SomeSeries:string) [
2    100, "Apple",
3    200, "Banana",]
```

| | SomeInt | SomeSeries |
|---|---|---|
| > | 100 | Apple |
| > | 200 | Banana |

```
1  let TempData = datatable(id:int, name:string)
2  [1, "John",
3  2, "Jackson",
4  3, "Jennifer"];
5  TempData
```

| | id | name |
|---|---|---|
| > | 1 | John |
| > | 2 | Jackson |
| > | 3 | Jennifer |

# KQL Operators: Joins

**The join operator** works similarly to the join operator in SQL.

It allows us to merge the rows of 2 tables together, based on some key.

Similar to other query languages, we have a number of kinds of joins.

**By default**, join will use the inner unique type of join.

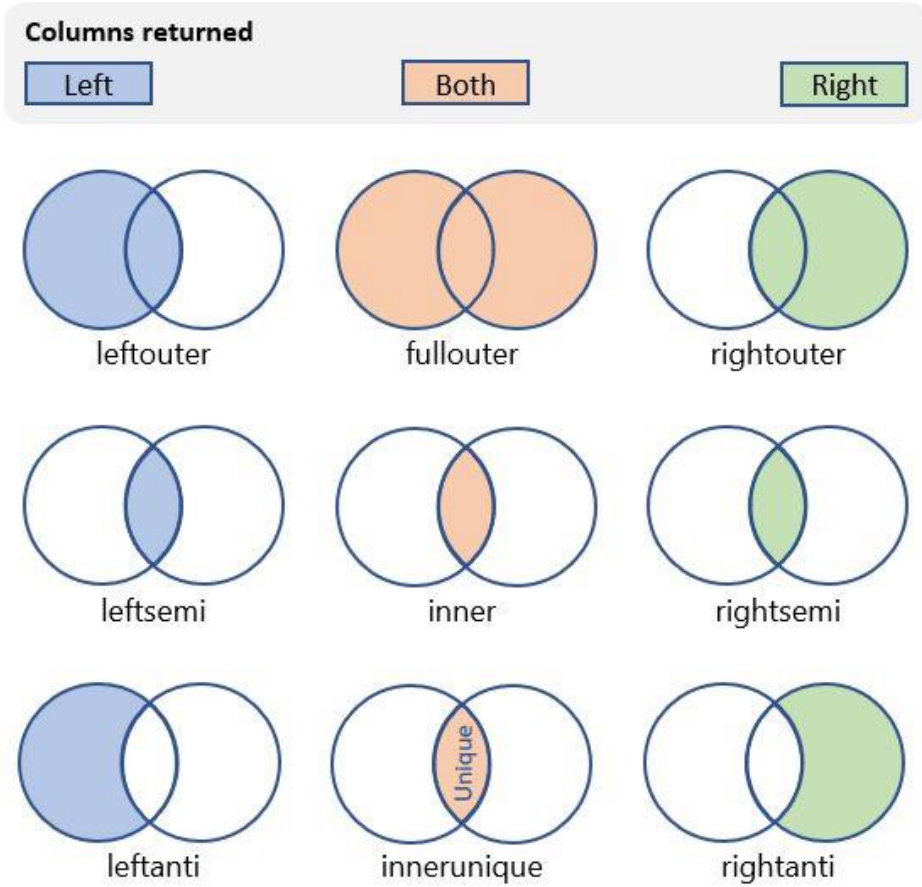
**Usage:** Table1 | join (Table2) on CommonColumn, $left.Col1 == $right.Col2

| Type of join | Performance |
|---|---|
| innerunique (or empty as default) | Inner join with left side deduplication |
| inner | Standard inner join |
| leftouter | Left outer join |
| rightouter | Right outer join |
| fullouter | Full outer join |
| leftanti, anti, or leftantisemi | Left anti join |
| rightanti or rightantisemi | Right anti join |
| leftsemi | Left semi join |
| rightsemi | Right semi join |

| SQL equivalent | KQL equivalent |
|---|---|
| `SELECT * FROM finances`<br>`LEFT OUTER JOIN exception`<br>`ON dependencies.operation_Id =`<br>`exceptions.operation_Id` | `finances`<br>`\| join kind = leftouter  (exceptions)`<br>`on $left.operation_Id == $right.operation_Id` |

# KQL Operators: Joins

- We can include multiple joins in a single query.
- Can use user-defined tables in our queries.
- KQL doesn't have a cross join, but we can get around this by assigning a dummy key.



# KQL Operators: Joins

- We can include multiple joins in a single query.
- Can use user-defined tables in our queries.
- KQL doesn't have a cross join, but we can get around this by assigning a dummy key.

## Join Examples

```
let Events = MyLogTable | where type=="Event" ;
Events
| where Name == "Start"
| project Name, City, ActivityId,
StartTime=timestamp
| join (Events
    | where Name == "Stop"
        | project StopTime=timestamp, ActivityId)
    on ActivityId
| project City, ActivityId, StartTime, StopTime,
Duration = StopTime - StartTime
```

```
let X = datatable(Key:string, Value1:long)
['a',1, 'b',2, 'b',3, 'c',4];
let Y = datatable(Key:string, Value2:long)
['b',10, 'c',20, 'c',30, 'd',40];
X | join kind=inner Y on Key
```

```
X | extend dk=1 | join kind=inner (Y | extend dk=1)
on dk
```

# KQL Operators: Unions

**The union operator** works similarly to unions operators in other query languages.

It will take two or more tables and returns the rows of all of them.

**Usage:** union [UnionParameters] [kind= inner|outer] [withsource=ColumnName] [isfuzzy= true|false] Table [, Table]...

| SQL equivalent | KQL equivalent |
| --- | --- |
| `SELECT * FROM finances`<br>`UNION`<br>`SELECT * FROM exceptions` | `union finances, exceptions` |
| `SELECT * FROM finances WHERE timestamp > ...`<br>`UNION`<br>`SELECT * FROM exceptions WHERE timestamp > ...` | `finances`<br>`\| where timestamp > ago(1d)`<br>`\| union (exceptions  \| where timestamp > ago(1d))` |

# KQL Statement: Let

**The let statement** allows us to assign variables equal to an expression or a function, or to create views.

**Let statements** allow us to break up complex queries into parts, define constants, and reuse variables.

Once assigned, the variable exists in memory and can be used through calls, joins, and other uses

**Usage:** let Name = ScalarExpressionOrTabularExpression

```
let x = 1;
```

```
let RecentLog = Logs
| where Timestamp > ago(1h);
```

```
let n = 10;
let place = "New York";
let cutoff = ago(93d);
Events
|where timestamp > cutoff
    and city == place
|take n
```

# KQL Statement: Set

**The set statement** allows us to assign query options for the duration of the query. These query options control how a query executes, runs, and what it returns.

For example, “set querytrace;” would enable query tracing for all queries after this one.

**Usage:** set *OptionName* [= *OptionValue]*

# KQL Batching

**Batching** in KQL allows a user to include multiple expression statements, delimited by ";", which will then return multiple tabular results.

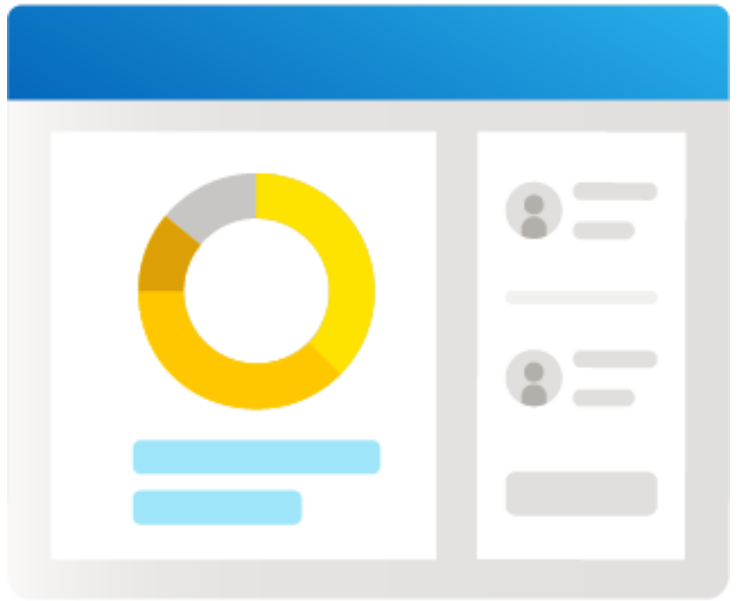
This is useful when a common calculation is shared across several subqueries, which will often occur in dashboards.

```
StormEvents
| project State, StartDate;

StormEvents
| where State == "New York"
| project State, EndDate
```

# Demonstration

Demo for Let, JOIN and Datatables



# Questions?

# Lesson 4: Advanced Functions.

# Objectives

After completing this Learning, you will be able to:

**1**

**Understand….**

- ✓ What is Pivot statement.
- ✓ Pivot statement use cases.

**2**

**Understand….**

- ✓ What is Basket statement.
- ✓ Basket statement use cases.

**3**

**Understand….**

- ✓ What is Autocluster statement.
- ✓ Autocluster statement use cases.

# Pivot statement

- Pivot statement rotates a table.
- Unique values of one column is converted into multiple columns of the output table.
- Aggregations on the remaining output columns.

```
StormEvents
| project State, EventType
| where State startswith "AL"
| where EventType has "Wind"
| evaluate pivot(State)
```

| EventType | ALABAMA | ALASKA |
|---|---|---|
| Thunderstorm Wind | 352 | 1 |
| High Wind | 0 | 95 |
| Extreme Cold/Wind Chill | 0 | 10 |
| Strong Wind | 22 | 0 |

# Basket statement

- Basket finds all frequent patterns of discrete attributes (dimensions) in the data.
- It returns the frequent patterns that passed the frequency threshold in the original query.
- Based on the Apriori algorithm originally developed for basket analysis data mining.

```
StormEvents
| where monthofyear(StartTime) == 5
| extend Damage = iff(DamageCrops +
DamageProperty > 0 , "YES" , "NO")
| project State, EventType, Damage, DamageCrops |
evaluate basket(0.2)
```

| SegmentId | Count | Percent | State | EventType | Damage | DamageCrops |
|---|---|---|---|---|---|---|
| 0 | 4574 | 77.7 | | | NO | 0 |
| 1 | 2278 | 38.7 | | Hail | NO | 0 |
| 2 | 5675 | 96.4 | | | | 0 |
| 3 | 2371 | 40.3 | | Hail | | 0 |

# Autocluster statement

- Autocluster finds common patterns of discrete attributes (dimensions) in the data.
- Reduces the results of the original query, whether it's 100 or 100k rows, to a small number of patterns.
- The plugin was developed to help analyze failures (such as exceptions or crashes).
- Can potentially work on any filtered data set.

```
StormEvents
| where monthofyear(StartTime) == 5
| extend Damage = iff(DamageCrops +
DamageProperty > 0 , "YES" , "NO")
| project State , EventType , Damage
| evaluate autocluster(0.6)
```

| SegmentId | Count | Percent | State | EventType | Damage |
|---|---|---|---|---|---|
| 0 | 2278 | 38.7 | | Hail | NO |
| 1 | 512 | 8.7 | | Thunderstorm Wind | YES |
| 2 | 898 | 15.3 | TEXAS | | |

# Questions?

# Resources

Continue your learning

1. aka.ms/adx.docs
2. aka.ms/adx.mslearn
3. aka.ms/adxinaday
4. detective.kusto.io
5. aka.ms/adx.blog
6. aka.ms/adx.youtube
7. aka.ms/adx.pluralsight

# Thank you