Scalar data types

Article • 03/28/2022

Every data value (such as the value of an expression, or the parameter to a function) has a **data type**. A data type is either a **scalar data type** (one of the built-in predefined types listed below), or a **user-defined record** (an ordered sequence of name/scalar-data-type pairs, such as the data type of a row of a table).

Kusto supplies a set of system data types that define all the types of data that can be used with Kusto.

> ⓘ Note
>
> User-defined data types are not supported in Kusto.

The following table lists the data types supported by Kusto, alongside additional aliases you can use to refer to them and a roughly equivalent .NET Framework type.

Type	Additional name(s)	Equivalent .NET type	gettype()
`bool`	`boolean`	`System.Boolean`	`int8`
`datetime`	`date`	`System.DateTime`	`datetime`
`dynamic`		`System.Object`	`array` or `dictionary` or any of the other values
`guid`		`System.Guid`	`guid`
`int`		`System.Int32`	`int`
`long`		`System.Int64`	`long`
`real`	`double`	`System.Double`	`real`
`string`		`System.String`	`string`
`timespan`	`time`	`System.TimeSpan`	`timespan`
`decimal`		`System.Data.SqlTypes.SqlDecimal`	`decimal`

All non-string data types include a special "null" value, which represents the lack of data or a mismatch of data. For example, attempting to ingest the string `"abc"` into an `int`

column results in this value. It isn't possible to materialize this value explicitly, but you can detect whether an expression evaluates to this value by using the `isnull()` function.

> ⚠ Warning
>
> Support for the `guid` type is incomplete. We strongly recommend that teams use values of type `string` instead.

Feedback

The bool data type

Article • 03/07/2022

The `bool` (`boolean`) data type can have one of two states: `true` or `false` (internally encoded as `1` and `0`, respectively), as well as the null value.

bool literals

The `bool` data type has the following literals:

- `true` and `bool(true)`: Representing trueness
- `false` and `bool(false)`: Representing falsehood
- `bool(null)`: See null values

bool operators

The `bool` data type supports the following operators: equality (`==`), inequality (`!=`), logical-and (`and`), and logical-or (`or`).

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

The datetime data type

Article • 08/24/2022

The `datetime` (`date`) data type represents an instant in time, typically expressed as a date and time of day. Values range from 00:00:00 (midnight), January 1, 0001 Anno Domini (Common Era) through 11:59:59 P.M., December 31, 9999 A.D. (C.E.) in the Gregorian calendar.

Time values are measured in 100-nanosecond units called ticks, and a particular date is the number of ticks since 12:00 midnight, January 1, 0001 A.D. (C.E.) in the GregorianCalendar calendar (excluding ticks that would be added by leap seconds). For example, a ticks value of 31241376000000000 represents the date, Friday, January 01, 0100 12:00:00 midnight. This is sometimes called "a moment in linear time".

> ⚠ Warning
>
> A `datetime` value in Kusto is always in the UTC time zone. If displaying `datetime` values in other time zones is required, please use **datetime_utc_to_local()** or its counterpart, **datetime_local_to_utc()**, to convert to a different time zone.

datetime literals

Literals of type `datetime` have the syntax `datetime(`*value*`)`, where a number of formats are supported for *value*, as indicated by the following table:

Example	Value
`datetime(2015-12-31 23:59:59.9)` `datetime(2015-12-31)`	Times are always in UTC. Omitting the date gives a time today.
`datetime(null)`	See null values.

The now() and ago() special functions

Kusto provides two special functions, now() and ago(), to allow queries to reference the time at which the query starts execution.

Supported formats

There are several formats for `datetime` that are supported as datetime() literals and the todatetime() function.

> ⚠ Warning
>
> It is **strongly recommended** to use only the ISO 8601 formats.

ISO 8601

Format	Example
%Y-%m-%dT%H:%M:%s%z	2014-05-25T08:20:03.123456Z
%Y-%m-%dT%H:%M:%s	2014-05-25T08:20:03.123456
%Y-%m-%dT%H:%M	2014-05-25T08:20
%Y-%m-%d %H:%M:%s%z	2014-11-08 15:55:55.123456Z
%Y-%m-%d %H:%M:%s	2014-11-08 15:55:55
%Y-%m-%d %H:%M	2014-11-08 15:55
%Y-%m-%d	2014-11-08

RFC 822

Format	Example
%w, %e %b %r %H:%M:%s %Z	Sat, 8 Nov 14 15:05:02 GMT
%w, %e %b %r %H:%M:%s	Sat, 8 Nov 14 15:05:02
%w, %e %b %r %H:%M	Sat, 8 Nov 14 15:05
%w, %e %b %r %H:%M %Z	Sat, 8 Nov 14 15:05 GMT
%e %b %r %H:%M:%s %Z	8 Nov 14 15:05:02 GMT
%e %b %r %H:%M:%s	8 Nov 14 15:05:02
%e %b %r %H:%M	8 Nov 14 15:05
%e %b %r %H:%M %Z	8 Nov 14 15:05 GMT

RFC 850

Format	Example
%w, %e-%b-%r %H:%M:%s %Z	Saturday, 08-Nov-14 15:05:02 GMT
%w, %e-%b-%r %H:%M:%s	Saturday, 08-Nov-14 15:05:02
%w, %e-%b-%r %H:%M %Z	Saturday, 08-Nov-14 15:05 GMT
%w, %e-%b-%r %H:%M	Saturday, 08-Nov-14 15:05
%e-%b-%r %H:%M:%s %Z	08-Nov-14 15:05:02 GMT
%e-%b-%r %H:%M:%s	08-Nov-14 15:05:02
%e-%b-%r %H:%M %Z	08-Nov-14 15:05 GMT
%e-%b-%r %H:%M	08-Nov-14 15:05

Sortable

Format	Example
%Y-%n-%e %H:%M:%s	2014-11-08 15:05:25
%Y-%n-%e %H:%M:%s %Z	2014-11-08 15:05:25 GMT
%Y-%n-%e %H:%M	2014-11-08 15:05
%Y-%n-%e %H:%M %Z	2014-11-08 15:05 GMT
%Y-%n-%eT%H:%M:%s	2014-11-08T15:05:25
%Y-%n-%eT%H:%M:%s %Z	2014-11-08T15:05:25 GMT
%Y-%n-%eT%H:%M	2014-11-08T15:05
%Y-%n-%eT%H:%M %Z	2014-11-08T15:05 GMT

Feedback

The decimal data type

Article • 03/06/2023

The `decimal` data type represents a 128-bit wide, decimal number.

Literals of the `decimal` data type have the same representation as .NET's `System.Data.SqlTypes.SqlDecimal`.

`decimal(1.0)`, `decimal(0.1)`, and `decimal(1e5)` are all literals of type `decimal`.

There are several special literal forms:

- `decimal(null)`: This is the null value.

> ⊗ **Caution**
>
> Arithmetic operations involving `decimal` values are significantly slower than operations on **real** data type. If your use case doesn't require very high precision, it's advised to switch to `real`.

Feedback

The dynamic data type

Article • 01/17/2023

The `dynamic` scalar data type is special in that it can take on any value of other scalar data types from the list below, as well as arrays and property bags. Specifically, a `dynamic` value can be:

- Null.
- A value of any of the primitive scalar data types: `bool`, `datetime`, `guid`, `int`, `long`, `real`, `string`, and `timespan`.
- An array of `dynamic` values, holding zero or more values with zero-based indexing.
- A property bag that maps unique `string` values to `dynamic` values. The property bag has zero or more such mappings (called "slots"), indexed by the unique `string` values. The slots are unordered.

> **Note**
>
> - Values of type `dynamic` are limited to 1MB (2^20), uncompressed.
> - Although the `dynamic` type appears JSON-like, it can hold values that the JSON model does not represent because they don't exist in JSON (e.g., `long`, `real`, `datetime`, `timespan`, and `guid`). Therefore, in serializing `dynamic` values into a JSON representation, values that JSON can't represent are serialized into `string` values. Conversely, Kusto will parse strings as strongly-typed values if they can be parsed as such. This applies for `datetime`, `real`, `long`, and `guid` types. For more about the JSON object model, see **json.org**.
> - Kusto doesn't attempt to preserve the order of name-to-value mappings in a property bag, and so you can't assume the order to be preserved. It's entirely possible for two property bags with the same set of mappings to yield different results when they are represented as `string` values, for example.

Dynamic literals

A literal of type `dynamic` looks like this:

`dynamic(` *Value* `)`

Value can be:

- `null`, in which case the literal represents the null dynamic value: `dynamic(null)`.
- Another scalar data type literal, in which case the literal represents the `dynamic` literal of the "inner" type. For example, `dynamic(4)` is a dynamic value holding the value 4 of the long scalar data type.
- An array of dynamic or other literals: `[` *ListOfValues* `]`. For example, `dynamic([1, 2, "hello"])` is a dynamic array of three elements, two `long` values and one `string` value.
- A property bag: `{` *Name* `=` *Value* ... `}`. For example, `dynamic({"a":1, "b":{"a":2}})` is a property bag with two slots, `a`, and `b`, with the second slot being another property bag.

Kusto

```
print o=dynamic({"a":123, "b":"hello", "c":[1,2,3], "d":{}})
| extend a=o.a, b=o.b, c=o.c, d=o.d
```

For convenience, `dynamic` literals that appear in the query text itself may also include other Kusto literals with types: `datetime`, `timespan`, `real`, `long`, `guid`, `bool`, and `dynamic`. This extension over JSON isn't available when parsing strings (such as when using the `parse_json` function or when ingesting data), but it enables you to do the following:

Kusto

```
print d=dynamic({"a": datetime(1970-05-11)})
```

To parse a `string` value that follows the JSON encoding rules into a `dynamic` value, use the `parse_json` function. For example:

- `parse_json('[43, 21, 65]')` - an array of numbers
- `parse_json('{"name":"Alan", "age":21, "address":{"street":432,"postcode":"JLK32P"}}')` - a dictionary
- `parse_json('21')` - a single value of dynamic type containing a number
- `parse_json('"21"')` - a single value of dynamic type containing a string
- `parse_json('{"a":123, "b":"hello", "c":[1,2,3], "d":{}}')` - gives the same value as `o` in the example above.

> ⓘ **Note**

Unlike JavaScript, JSON mandates the use of double-quote (`"`) characters around strings and property-bag property names. Therefore, it is generally easier to quote a JSON-encoded string literal by using a single-quote (`'`) character.

The following example shows how you can define a table that holds a `dynamic` column (as well as a `datetime` column) and then ingest into it a single record. it also demonstrates how you can encode JSON strings in CSV files:

Kusto

```
// dynamic is just like any other type:
.create table Logs (Timestamp:datetime, Trace:dynamic)

// Everything between the "[" and "]" is parsed as a CSV line would be:
// 1. Since the JSON string includes double-quotes and commas (two
characters
//    that have a special meaning in CSV), we must CSV-quote the entire
second field.
// 2. CSV-quoting means adding double-quotes (") at the immediate beginning
and end
//    of the field (no spaces allowed before the first double-quote or after
the second
//    double-quote!)
// 3. CSV-quoting also means doubling-up every instance of a double-quotes
within
//    the contents.
.ingest inline into table Logs
  [2015-01-01,"{""EventType"":""Demo"", ""EventValue"":""Double-quote
love!""}"]
```

Output

Timestamp	Trace
2015-01-01 00:00:00.0000000	{"EventType":"Demo","EventValue":"Double-quote love!"}

Dynamic object accessors

To subscript a dictionary, use either the dot notation (`dict.key`) or the brackets notation (`dict["key"]`). When the subscript is a string constant, both options are equivalent.

> ⓘ **Note**
>
> To use an expression as the subscript, use the brackets notation. When using arithmetic expressions, the expression inside the brackets must be wrapped in

parentheses.

In the examples below `dict` and `arr` are columns of dynamic type:

Expression	Accessor expression type	Meaning	Comments
dict[col]	Entity name (column)	Subscripts a dictionary using the values of the column `col` as the key	Column must be of type string
arr[index]	Entity index (column)	Subscripts an array using the values of the column `index` as the index	Column must be of type integer or boolean
arr[-index]	Entity index (column)	Retrieves the 'index'-th value from the end of the array	Column must be of type integer or boolean
arr[(-1)]	Entity index	Retrieves the last value in the array	
arr[toint(indexAsString)]	Function call	Casts the values of column `indexAsString` to int and use them to subscript an array	
dict[['where']]	Keyword used as entity name (column)	Subscripts a dictionary using the values of column `where` as the key	Entity names that are identical to some query language keywords must be quoted
dict.['where'] or dict['where']	Constant	Subscripts a dictionary using `where` string as the key	

Performance tip: Prefer to use constant subscripts when possible

Accessing a sub-object of a `dynamic` value yields another `dynamic` value, even if the sub-object has a different underlying type. Use the `gettype` function to discover the actual underlying type of the value, and any of the cast function listed below to cast it to the actual type.

Casting dynamic objects

> After subscripting a dynamic object, you must cast the value to a simple type.

Expression	Value	Type
X	parse_json('[100,101,102]')	array
X[0]	parse_json('100')	dynamic
toint(X[1])	101	int
Y	parse_json('{"a1":100, "a b c":"2015-01-01"}')	dictionary
Y.a1	parse_json('100')	dynamic
Y["a b c"]	parse_json("2015-01-01")	dynamic
todate(Y["a b c"])	datetime(2015-01-01)	datetime

Cast functions are:

- `tolong()`
- `todouble()`
- `todatetime()`
- `totimespan()`
- `tostring()`
- `toguid()`
- `parse_json()`

Building dynamic objects

Several functions enable you to create new `dynamic` objects:

- bag_pack() creates a property bag from name/value pairs.
- pack_array() creates an array from name/value pairs.
- range() creates an array with an arithmetic series of numbers.
- zip() pairs "parallel" values from two arrays into a single array.
- repeat() creates an array with a repeated value.

Additionally, there are several aggregate functions which create `dynamic` arrays to hold aggregated values:

- buildschema() returns the aggregate schema of multiple `dynamic` values.
- make_bag() returns a property bag of dynamic values within the group.

- make_bag_if() returns a property bag of dynamic values within the group (with a predicate).
- make_list() returns an array holding all values, in sequence.
- make_list_if() returns an array holding all values, in sequence (with a predicate).
- make_list_with_nulls() returns an array holding all values, in sequence, including null values.
- make_set() returns an array holding all unique values.
- make_set_if() returns an array holding all unique values (with a predicate).

Operators and functions over dynamic types

For a complete list of scalar dynamic/array functions, see dynamic/array functions.

Operator or function	Usage with dynamic data types
value `in` *array*	True if there's an element of *array* that == *value* `where City in ('London', 'Paris', 'Rome')`
value `!in` *array*	True if there's no element of *array* that == *value*
array_length(array)	Null if it isn't an array
bag_has_key(bag,key)	Checks whether a dynamic bag column contains a given key.
bag_keys(bag)	Enumerates all the root keys in a dynamic property-bag object.
bag_merge(bag1,...,bagN)	Merges dynamic property-bags into a dynamic property-bag with all properties merged.
bag_set_key(bag,key,value)	Sets a given key to a given value in a dynamic property-bag.
extract_json(path,object), extract_json(path,object)	Use path to navigate into object.
parse_json(source)	Turns a JSON string into a dynamic object.
range(from,to,step)	An array of values
mv-expand listColumn	Replicates a row for each value in a list in a specified cell.
summarize buildschema(column)	Infers the type schema from column content
summarize make_bag(column)	Merges the property bag (dictionary) values in the column into one property bag, without key duplication.
summarize make_bag_if(column,predicate)	Merges the property bag (dictionary) values in the column into one property bag, without key duplication (with predicate).

Operator or function	Usage with dynamic data types
summarize make_list(column)	Flattens groups of rows and puts the values of the column in an array.
summarize make_list_if(column,predicate)	Flattens groups of rows and puts the values of the column in an array (with predicate).
summarize make_list_with_nulls(column)	Flattens groups of rows and puts the values of the column in an array, including null values.
summarize make_set(column)	Flattens groups of rows and puts the values of the column in an array, without duplication.

Indexing for dynamic data

Every field is indexed during data ingestion. The scope of the index is a single data shard.

To index dynamic columns, the ingestion process enumerates all "atomic" elements within the dynamic value (property names, values, array elements) and forwards them to the index builder. Otherwise, dynamic fields have the same inverted term index as string fields.

Next steps

- To see an example query using dynamic objects and object accessors, see Map values from one set to another.

Feedback

The guid data type

Article • 06/15/2023

The `guid` (`uuid`, `uniqueid`) data type represents a 128-bit globally-unique value.

guid literals

To represent a literal of type `guid`, use the following format:

Kusto

```
guid(74be27de-1e4e-49d9-b579-fe0b331d3642)
```

The special form `guid(null)` represents the null value.

Feedback

The int data type

Article • 03/07/2022

The `int` data type represents a signed, 32-bit wide, integer.

The special form `int(null)` represents the null value.

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

The long data type

Article • 03/07/2022

The `long` data type represents a signed, 64-bit wide, integer.

long literals

Literals of the `long` data type can be specified in the following syntax:

`long` `(` *Value* `)`

Where *Value* can take the following forms:

- One more or digits, in which case the literal value is the decimal representation of these digits. For example, `long(12)` is the number twelve of type `long`.
- The prefix `0x` followed by one or more Hex digits. For example, `long(0xf)` is equivalent to `long(15)`.
- A minus (`-`) sign followed by one or more digits. For example, `long(-1)` is the number minus one of type `long`.
- `null`, in which case this is the null value of the `long` data type. Thus, the null value of type `long` is `long(null)`.

Kusto also supports literals of type `long` without the `long(`/`)` prefix/suffi for the first two forms only. Thus, `123` is a literal of type `long`, as is `0x123`, but `-2` is **not** a literal (it is currently interpreted as the unary function `-` applied to the literal `2` of type long).

For converting long into hex string - see tohex() function.

Feedback

The real data type

Article • 03/07/2022

The `real` data type represents a 64-bit wide, double-precision, floating-point number.

Literals of the `real` data type have the same representation as .NET's `System.Double`. `1.0`, `0.1`, and `1e5` are all literals of type `real`.

There are several special literal forms:

- `real(null)`: The is the null value.
- `real(nan)`: Not-a-Number (NaN). For example, the result of dividing a `0.0` by another `0.0`.
- `real(+inf)`: Positive infinity. For example, the result of dividing `1.0` by `0.0`.
- `real(-inf)`: Negative infinity. For example, the result of dividing `-1.0` by `0.0`.

Feedback

The string data type

Article • 03/15/2023

The `string` data type represents a sequence of zero or more Unicode characters.

> **Note**
>
> - Internally, strings are encoded in **UTF-8**. Invalid (non-UTF8) characters are replaced with **U+FFFD** Unicode replacement characters at ingestion time.
> - Kusto has no data type that is equivalent to a single character. A single character is represented as a string of length 1.
> - When ingesting the `string` data type, if a single string value in a record exceeds 1MB (measured using UTF-8 encoding), the value is truncated and ingestion succeeds. If a single string value in a record, or the entire record, exceeds the allowed data limit of 64MB, ingestion fails.

String literals

There are several ways to encode literals of the `string` data type in a query text:

- Enclose the string in double-quotes (`"`): `"This is a string literal. Single quote characters (') don't require escaping. Double quote characters (") are escaped by a backslash (\)."`
- Enclose the string in single-quotes (`'`): `'Another string literal. Single quote characters (') require escaping by a backslash (\). Double quote characters (") do not require escaping.'`

In the two representations above, the backslash (`\`) character indicates escaping. The backslash is used to escape the enclosing quote characters, tab characters (`\t`), newline characters (`\n`), and itself (`\\`).

> **Note**
>
> The newline character (`\n`) and the return character (`\r`) can't be included as part of the string literal without being quoted. See also **multi-line string literals**.

Verbatim string literals

Verbatim string literals are also supported. In this form, the backslash character (`\`) stands for itself, and not as an escape character. Prepending the `@` special character to string literals serves as a verbatim identifier.

- Enclose in double-quotes (`"`): `@"This is a verbatim string literal that ends with a backslash\. Double quote characters (") are escaped by a double quote (")."`
- Enclose in single-quotes (`'`): `@'This is a verbatim string literal that ends with a backslash\. Single quote characters (') are escaped by a single quote (')."`

> ⓘ **Note**
>
> The newline character (`\n`) and the return character (`\r`) can't be included as part of the string literal without being quoted. See also **multi-line string literals**.

Splicing string literals

Two or more string literals are automatically joined to form a new string literal in the query if they have nothing between them, or they're separated only by whitespace and comments.
For example, the following expressions all yield a string of length 13:

Kusto

```
print strlen("Hello"', '@"world!"); // Nothing between them

print strlen("Hello" ', ' @"world!"); // Separated by whitespace only

print strlen("Hello"
  // Comment
  ', '@"world!"); // Separated by whitespace and a comment
```

Multi-line string literals

Multi-line string literals are string literals for which the newline (`\n`) and return (`\r`) characters don't require escaping.

- Multi-line string literals always appear between two occurrences of the "triple-backtick chord" (```).

> ⓘ Note
>
> - Multi-line string literals do not support escaped characters. Similar to **verbatim string literals**, multi-line string literals allow newline and return characters.
> - Multi-line string literals don't support obfuscation.

Examples

Kusto

```
// Simple string notation
print s1 = 'some string', s2 = "some other string"

// Strings that include single or double-quotes can be defined as follows
print s1 = 'string with " (double quotes)',
          s2 = "string with ' (single quotes)"

// Strings with '\' can be prefixed with '@' (as in c#)
print myPath1 = @'C:\Folder\filename.txt'

// Escaping using '\' notation
print s = '\\n.*(>|\'|=|\")[a-zA-Z0-9/+]{86}=='

// Encode a C# program in a Kusto multi-line string
print program=```
  public class Program {
    public static void Main() {
      System.Console.WriteLine("Hello!");
    }
  }```
```

As can be seen, when a string is enclosed in double-quotes (`"`), the single-quote (`'`) character doesn't require escaping, and also the other way around. This method makes it easier to quote strings according to context.

Obfuscated string literals

The system tracks queries and stores them for telemetry and analysis purposes. For example, the query text might be made available to the cluster owner. If the query text includes secret information, such as passwords, it might leak information that should be kept private. To prevent such a leak from happening, the query author may mark specific string literals as **obfuscated string literals**. Such literals in the query text are

automatically replaced by a number of star (`*`) characters, so that they aren't available for later analysis.

> **Important**
>
> Mark all string literals that contain secret information, as obfuscated string literals.

An obfuscated string literal can be formed by taking a "regular" string literal, and prepending an `h` or an `H` character in front of it.

For example:

Kusto

```
h'hello'
h@'world'
h"hello"
```

> **Note**
>
> In many cases, only a part of the string literal is secret. In those cases, split the literal into a non-secret part and a secret part. Then, only mark the secret part as obfuscated.

For example:

Kusto

```
print x="https://contoso.blob.core.windows.net/container/blob.txt?"
  h'sv=2012-02-12&se=2013-04-13T0...'
```

Feedback

The timespan data type

Article • 03/07/2022

The `timespan` (`time`) data type represents a time interval.

timespan literals

Literals of type `timespan` have the syntax `timespan(`*value*`)`, where a number of formats are supported for *value*, as indicated by the following table:

Value	Length of time
`2d`	2 days
`1.5h`	1.5 hour
`30m`	30 minutes
`10s`	10 seconds
`0.1s`	0.1 second
`100ms`	100 millisecond
`10microsecond`	10 microseconds
`1tick`	100ns
`time(15 seconds)`	15 seconds
`time(2)`	2 days
`time(0.12:34:56.7)`	`0d+12h+34m+56.7s`

The special form `time(null)` is the null value.

timespan operators

Two values of type `timespan` may be added, subtracted, and divided. The last operation returns a value of type `real` representing the fractional number of times one value can fit the other.

Examples

The following example calculates how many seconds are in a day in several ways:

Kusto

```
print
    result1 = 1d / 1s,
    result2 = time(1d) / time(1s),
    result3 = 24 * 60 * time(00:01:00) / time(1s)
```

This example converts the number of seconds in a day (represented by an integer value) to a timespan unit:

Kusto

```
print
    seconds = 86400
| extend t = seconds * 1s
```

Feedback

Null Values

Article • 01/16/2023

All scalar data types in Kusto have a special value that represents a missing value. This value is called the **null value**, or **null**.

> ⓘ Note
>
> The `string` data type doesn't support null values.

Null literals

The null value of a scalar type *T* is represented in the query language by the null literal `T(null)`. The following query returns a single row full of null values:

Kusto

```
print bool(null), datetime(null), dynamic(null), guid(null), int(null),
long(null), real(null), double(null), time(null)
```

Predicates on null values

The scalar function isnull() can be used to determine if a scalar value is the null value. The corresponding function isnotnull() can be used to determine if a scalar value isn't the null value.

> ⓘ Note
>
> Because the `string` type doesn't support null values, it's recommended to use the **isempty()** and the **isnotempty()** functions.

Equality and inequality of null values

- **Equality** (`==`): Applying the equality operator to two null values yields `bool(null)`. Applying the equality operator to a null value and a non-null value yields `bool(false)`.

- **Inequality** (`!=`): Applying the inequality operator to two null values yields `bool(null)`. Applying the inequality operator to a null value and a non-null value yields `bool(true)`.

For example:

```
Kusto

datatable(val:int)[5, int(null)]
| extend IsBiggerThan3 = val > 3
| extend IsBiggerThan3OrNull = val > 3 or isnull(val)
| extend IsEqualToNull = val == int(null)
| extend IsNotEqualToNull = val != int(null)
```

Results:

val	IsBiggerThan3	IsBiggerThan3OrNull	IsEqualToNull	IsNotEqualToNull
5	true	true	false	true
null	null	true	null	null

Null values and the where query operator

The where operator use Boolean expressions to determine if to emit each input record to the output. This operator treats null values as if they're `bool(false)`. Records for which the predicate returns the null value are dropped and don't appear in the output.

For example:

```
Kusto

datatable(ival:int, sval:string)[5, "a", int(null), "b"]
| where ival != 5
```

Results:

ival	sval
null	b

Binary operators and null values

Binary operators are scalar operators that accept two scalar values and produce a third value. For example, greater-than (>) and Boolean AND (&&) are binary operators.

For all binary operators except as noted below, the rule is as follows:
If one or both of the values input to the binary operator are null values, then the output of the binary operator is also the null value. In other words, the null value is "sticky".

Exceptions to this rule

- For the equality (`==`) and inequality (`!=`) operators, if one of the values is null and the other value isn't null, then the result is either `bool(false)` or `bool(true)`, respectively.
- For the logical AND (&&) operator, if one of the values is `bool(false)`, the result is also `bool(false)`.
- For the logical OR (`||`) operator, if one of the values is `bool(true)`, the result is also `bool(true)`.

For example:

Kusto

```
datatable(val:int)[5, int(null)]
| extend Add = val + 10
| extend Multiply = val * 10
```

Results:

val	Add	Multiply
5	15	50
null	null	null

Null values and the logical NOT operator

The logical NOT operator not() yields the value `bool(null)` if the argument is the null value.

Null values and the in operator

- The in operator behaves like a logical OR of equality comparisons.
- The !in operator behaves like a logical AND of inequality comparisons.

Data ingestion and null values

For most data types, a missing value in the data source produces a null value in the corresponding table cell. However, columns of type `string` and CSV (or CSV-like) data formats are an exception to this rule, and a missing value produces an empty string.

For example:

Kusto

```
.create table T(a:string, b:int)

.ingest inline into table T
[,]
[ , ]
[a,1]
```

Output

a	b	isnull(a)	isempty(a)	strlen(a)	isnull(b)
		false	true	0	true
		false	false	1	true
a	1	false	false	1	false

> **Note**
>
> - If you run the above query in Kusto.Explorer, all `true` values will be displayed as `1`, and all `false` values will be displayed as `0`.
> - Kusto doesn't offer a way to constrain a table's column from having null values. In other words, there's no equivalent to SQL's `NOT NULL` constraint.

Feedback

Unsupported scalar data types

Article • 03/07/2022

All undocumented **scalar data type** are considered unsupported in Kusto.

Among the unsupported types are the following. Some types were previously supported:

Type	Additional name(s)	Equivalent .NET type	Storage Type (internal name)
`float`		`System.Single`	`R32`
`int16`		`System.Int16`	`I16`
`uint16`		`System.UInt16`	`UI16`
`uint32`	`uint`	`System.UInt32`	`UI32`
`uint64`	`ulong`	`System.UInt64`	`UI64`
`uint8`	`byte`	`System.Byte`	`UI8`

Feedback