format_datetime()

Article • 12/28/2022

Formats a datetime according to the provided format.

Syntax

`format_datetime(`*date* `,` *format*`)`

Parameters

Name	Type	Required	Description
date	datetime	✓	The value to format.
format	string	✓	The output format comprised of one or more of the supported format elements.

Supported format elements

The *format* parameter should include one or more of the following elements:

Format specifier	Description	Examples
`d`	The day of the month, from 1 through 31.	2009-06-01T13:45:30 -> 1, 2009-06-15T13:45:30 -> 15
`dd`	The day of the month, from 01 through 31.	2009-06-01T13:45:30 -> 01, 2009-06-15T13:45:30 -> 15
`f`	The tenths of a second in a date and time value.	2009-06-15T13:45:30.6170000 -> 6, 2009-06-15T13:45:30.05 -> 0
`ff`	The hundredths of a second in a date and time value.	2009-06-15T13:45:30.6170000 -> 61, 2009-06-15T13:45:30.0050000 -> 00
`fff`	The milliseconds in a date and time value.	6/15/2009 13:45:30.617 -> 617, 6/15/2009 13:45:30.0005 -> 000
`ffff`	The ten thousandths of a second in a date and time value.	2009-06-15T13:45:30.6175000 -> 6175, 2009-06-15T13:45:30.0000500 -> 0000

Format specifier	Description	Examples
`fffff`	The hundred thousandths of a second in a date and time value.	2009-06-15T13:45:30.6175400 -> 61754, 2009-06-15T13:45:30.000005 -> 00000
`ffffff`	The millionths of a second in a date and time value.	2009-06-15T13:45:30.6175420 -> 617542, 2009-06-15T13:45:30.0000005 -> 000000
`fffffff`	The ten millionths of a second in a date and time value.	2009-06-15T13:45:30.6175425 -> 6175425, 2009-06-15T13:45:30.0001150 -> 0001150
`F`	If non-zero, the tenths of a second in a date and time value.	2009-06-15T13:45:30.6170000 -> 6, 2009-06-15T13:45:30.0500000 -> (no output)
`FF`	If non-zero, the hundredths of a second in a date and time value.	2009-06-15T13:45:30.6170000 -> 61, 2009-06-15T13:45:30.0050000 -> (no output)
`FFF`	If non-zero, the milliseconds in a date and time value.	2009-06-15T13:45:30.6170000 -> 617, 2009-06-15T13:45:30.0005000 -> (no output)
`FFFF`	If non-zero, the ten thousandths of a second in a date and time value.	2009-06-15T13:45:30.5275000 -> 5275, 2009-06-15T13:45:30.0000500 -> (no output)
`FFFFF`	If non-zero, the hundred thousandths of a second in a date and time value.	2009-06-15T13:45:30.6175400 -> 61754, 2009-06-15T13:45:30.0000050 -> (no output)
`FFFFFF`	If non-zero, the millionths of a second in a date and time value.	2009-06-15T13:45:30.6175420 -> 617542, 2009-06-15T13:45:30.0000005 -> (no output)
`FFFFFFF`	If non-zero, the ten millionths of a second in a date and time value.	2009-06-15T13:45:30.6175425 -> 6175425, 2009-06-15T13:45:30.0001150 -> 000115
`h`	The hour, using a 12-hour clock from 1 to 12.	2009-06-15T01:45:30 -> 1, 2009-06-15T13:45:30 -> 1
`hh`	The hour, using a 12-hour clock from 01 to 12.	2009-06-15T01:45:30 -> 01, 2009-06-15T13:45:30 -> 01
`H`	The hour, using a 24-hour clock from 0 to 23.	2009-06-15T01:45:30 -> 1, 2009-06-15T13:45:30 -> 13

Format specifier	Description	Examples
`HH`	The hour, using a 24-hour clock from 00 to 23.	2009-06-15T01:45:30 -> 01, 2009-06-15T13:45:30 -> 13
`m`	The minute, from 0 through 59.	2009-06-15T01:09:30 -> 9, 2009-06-15T13:29:30 -> 29
`mm`	The minute, from 00 through 59.	2009-06-15T01:09:30 -> 09, 2009-06-15T01:45:30 -> 45
`M`	The month, from 1 through 12.	2009-06-15T13:45:30 -> 6
`MM`	The month, from 01 through 12.	2009-06-15T13:45:30 -> 06
`s`	The second, from 0 through 59.	2009-06-15T13:45:09 -> 9
`ss`	The second, from 00 through 59.	2009-06-15T13:45:09 -> 09
`y`	The year, from 0 to 99.	0001-01-01T00:00:00 -> 1, 0900-01-01T00:00:00 -> 0, 1900-01-01T00:00:00 -> 0, 2009-06-15T13:45:30 -> 9, 2019-06-15T13:45:30 -> 19
`yy`	The year, from 00 to 99.	0001-01-01T00:00:00 -> 01, 0900-01-01T00:00:00 -> 00, 1900-01-01T00:00:00 -> 00, 2019-06-15T13:45:30 -> 19
`yyyy`	The year as a four-digit number.	0001-01-01T00:00:00 -> 0001, 0900-01-01T00:00:00 -> 0900, 1900-01-01T00:00:00 -> 1900, 2009-06-15T13:45:30 -> 2009
`tt`	AM / PM hours	2009-06-15T13:45:09 -> PM

Supported delimiters

The format specifier can include the following delimiters:

Delimiter	Comment
`' '`	Space
`'/'`	
`'-'`	Dash

Delimiter	Comment
`':'`	
`','`	
`'.'`	
`'_'`	
`'['`	
`']'`	

Returns

A string with *date* formatted as specified by *format*.

Examples

Run the query

Kusto

```
let dt = datetime(2017-01-29 09:00:05);
print
v1=format_datetime(dt,'yy-MM-dd [HH:mm:ss]'),
v2=format_datetime(dt, 'yyyy-M-dd [H:mm:ss]'),
v3=format_datetime(dt, 'yy-MM-dd [hh:mm:ss tt]')
```

Output

v1	v2	v3
17-01-29 [09:00:05]	2017-1-29 [9:00:05]	17-01-29 [09:00:05 AM]

Feedback

format_ipv4()

Article • 12/28/2022

Parses the input with a netmask and returns a string representing the IPv4 address.

Syntax

`format_ipv4(`*ip* `[,` *prefix*`])`

Parameters

Name	Type	Required	Description
ip	string	✓	The IPv4 address. The format may be a string or number representation in big-endian order.
prefix	int		An integer from 0 to 32 representing the number of most-significant bits that are taken into account. If unspecified, all 32 bit-masks are used.

Returns

If conversion is successful, the result will be a string representing IPv4 address. If conversion isn't successful, the result will be an empty string.

Examples

Run the query

Kusto

```
datatable(address:string, mask:long)
[
 '192.168.1.1', 24,
 '192.168.1.1', 32,
 '192.168.1.1/24', 32,
 '192.168.1.1/24', long(-1),
]
| extend result = format_ipv4(address, mask),
         result_mask = format_ipv4_mask(address, mask)
```

Output

address	mask	result	result_mask
192.168.1.1	24	192.168.1.0	192.168.1.0/24
192.168.1.1	32	192.168.1.1	192.168.1.1/32
192.168.1.1/24	32	192.168.1.0	192.168.1.0/24
192.168.1.1/24	-1		

See also

- For IPv4 address formatting including CIDR notation, see format_ipv4_mask().
- For a list of functions related to IP addresses, see IPv4 and IPv6 functions.

Feedback

format_ipv4_mask()

Article • 12/28/2022

Parses the input with a netmask and returns a string representing the IPv4 address in CIDR notation.

Syntax

`format_ipv4_mask(`*ip* `[,` *prefix*`])`

Parameters

Name	Type	Required	Description
ip	string	✓	The IPv4 address as CIDR notation. The format may be a string or number representation in big-endian order.
prefix	int		An integer from 0 to 32 representing the number of most-significant bits that are taken into account. If unspecified, all 32 bit-masks are used.

Returns

If conversion is successful, the result will be a string representing IPv4 address as CIDR notation. If conversion isn't successful, the result will be an empty string.

Examples

Run the query

Kusto

```
datatable(address:string, mask:long)
[
 '192.168.1.1', 24,
 '192.168.1.1', 32,
 '192.168.1.1/24', 32,
 '192.168.1.1/24', long(-1),
]
| extend result = format_ipv4(address, mask),
         result_mask = format_ipv4_mask(address, mask)
```

Output

address	mask	result	result_mask
192.168.1.1	24	192.168.1.0	192.168.1.0/24
192.168.1.1	32	192.168.1.1	192.168.1.1/32
192.168.1.1/24	32	192.168.1.0	192.168.1.0/24
192.168.1.1/24	-1		

See also

- For IPv4 address formatting without CIDR notation, see format_ipv4().
- For a list of functions related to IP addresses, see IPv4 and IPv6 functions.

Feedback

format_timespan()

Article • 12/28/2022

Formats a timespan according to the provided format.

Syntax

`format_timespan(`*timespan* `,` *format*`)`

Name	Type	Required	Description
timespan	timespan	✓	The value to format.
format	string	✓	The output format comprised of one or more of the supported format elements.

Supported format elements

Format specifier	Description	Examples
`d`-`ddddddd`	The number of whole days in the time interval. Padded with zeros if needed.	15.13:45:30: d -> 15, dd -> 15, ddd -> 015
`f`	The tenths of a second in the time interval.	15.13:45:30.6170000 -> 6, 15.13:45:30.05 -> 0
`ff`	The hundredths of a second in the time interval.	15.13:45:30.6170000 -> 61, 15.13:45:30.0050000 -> 00
`fff`	The milliseconds in the time interval.	6/15/2009 13:45:30.617 -> 617, 6/15/2009 13:45:30.0005 -> 000
`ffff`	The ten thousandths of a second in the time interval.	15.13:45:30.6175000 -> 6175, 15.13:45:30.0000500 -> 0000
`fffff`	The hundred thousandths of a second in the time interval.	15.13:45:30.6175400 -> 61754, 15.13:45:30.000005 -> 00000

Format specifier	Description	Examples
`ffffff`	The millionths of a second in the time interval.	15.13:45:30.6175420 -> 617542, 15.13:45:30.0000005 -> 000000
`fffffff`	The ten millionths of a second in the time interval.	15.13:45:30.6175425 -> 6175425, 15.13:45:30.0001150 -> 0001150
`F`	If non-zero, the tenths of a second in the time interval.	15.13:45:30.6170000 -> 6, 15.13:45:30.0500000 -> (no output)
`FF`	If non-zero, the hundredths of a second in the time interval.	15.13:45:30.6170000 -> 61, 15.13:45:30.0050000 -> (no output)
`FFF`	If non-zero, the milliseconds in the time interval.	15.13:45:30.6170000 -> 617, 15.13:45:30.0005000 -> (no output)
`FFFF`	If non-zero, the ten thousandths of a second in the time interval.	15.13:45:30.5275000 -> 5275, 15.13:45:30.0000500 -> (no output)
`FFFFF`	If non-zero, the hundred thousandths of a second in the time interval.	15.13:45:30.6175400 -> 61754, 15.13:45:30.0000050 -> (no output)
`FFFFFF`	If non-zero, the millionths of a second in the time interval.	15.13:45:30.6175420 -> 617542, 15.13:45:30.0000005 -> (no output)
`FFFFFFF`	If non-zero, the ten millionths of a second in the time interval.	15.13:45:30.6175425 -> 6175425, 15.13:45:30.0001150 -> 000115
`H`	The hour, using a 24-hour clock from 0 to 23.	15.01:45:30 -> 1, 15.13:45:30 -> 13
`HH`	The hour, using a 24-hour clock from 00 to 23.	15.01:45:30 -> 01, 15.13:45:30 -> 13

Format specifier	Description	Examples
`m`	The number of whole minutes in the time interval that aren't included as part of hours or days. Single-digit minutes don't have a leading zero.	15.01:09:30 -> 9, 15.13:29:30 -> 29
`mm`	The number of whole minutes in the time interval that aren't included as part of hours or days. Single-digit minutes have a leading zero.	15.01:09:30 -> 09, 15.01:45:30 -> 45
`s`	The number of whole seconds in the time interval that aren't included as part of hours, days, or minutes. Single-digit seconds don't have a leading zero.	15.13:45:09 -> 9
`ss`	The number of whole seconds in the time interval that aren't included as part of hours, days, or minutes. Single-digit seconds have a leading zero.	15.13:45:09 -> 09

Supported delimiters

The format specifier can include following delimiters:

Delimiter	Comment
`' '`	Space
`'/'`	
`'-'`	Dash
`':'`	
`','`	
`'.'`	
`'_'`	
`'['`	
`']'`	

Returns

A string with *timespan* formatted as specified by *format*.

Examples

Run the query

Kusto

```
let t = time(29.09:00:05.12345);
print
v1=format_timespan(t, 'dd.hh:mm:ss:FF'),
v2=format_timespan(t, 'ddd.h:mm:ss [fffffff]')
```

Output

v1	v2
29.09:00:05:12	029.9:00:05 [1234500]

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

gamma()

Article • 12/27/2022

Computes the gamma function for the provided *number*.

Syntax

`gamma(`*number*`)`

Parameters

Name	Type	Required	Description
number	real	✓	The number used to calculate the gamma function.

Returns

Gamma function of *number*.

See also

For computing log-gamma function, see loggamma().

Feedback

geo_info_from_ip_address()

Article • 05/23/2023

Retrieves geolocation information about IPv4 or IPv6 addresses.

Syntax

`geo_info_from_ip_address(`*IpAddress*`)`

Parameters

Name	Type	Required	Description
IpAddress	string	✓	IPv4 or IPv6 address to retrieve geolocation information about.

Returns

A dynamic object containing the information on IP address whereabouts (if the information is available). The object contains the following fields:

Name	Type	Description
`country`	string	Country name
`state`	string	State (subdivision) name
`city`	string	City name
`latitude`	real	Latitude coordinate
`longitude`	real	Longitude coordinate

> ⓘ Note
>
> - IP geolocation is inherently imprecise; locations are often near the center of the population. Any location provided by this function should not be used to identify a particular address or household.
> - This function uses GeoLite2 data created by MaxMind, available from **https://www.maxmind.com**.

- The function is also built on the MaxMind DB Reader library provided under ISC license.

Examples

Run the query

```
Kusto

print ip_location=geo_info_from_ip_address('20.53.203.50')
```

Output

ip_location
{"country": "Australia", "state": "New South Wales", "city": "Sydney", "latitude": -33.8715, "longitude": 151.2006}

Run the query

```
Kusto

print
ip_location=geo_info_from_ip_address('2a03:2880:f12c:83:face:b00c::25de')
```

Output

ip_location
{"country": "United States", "state": "Florida", "city": "Boca Raton", "latitude": 26.3594, "longitude": -80.0771}

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

gettype()

Article • 01/02/2023

Returns the runtime type of its single argument.

The runtime type may be different than the nominal (static) type for expressions whose nominal type is `dynamic`; in such cases `gettype()` can be useful to reveal the type of the actual value (how the value is encoded in memory).

Syntax

`gettype(`*value*`)`

Parameters

Name	Type	Required	Description
value	scalar	✓	The value for which to find the type.

Returns

A string representing the runtime type of *value*.

Examples

Expression	Returns
`gettype("a")`	`string`
`gettype(111)`	`long`
`gettype(1==1)`	`bool`
`gettype(now())`	`datetime`
`gettype(1s)`	`timespan`
`gettype(parse_json('1'))`	`int`
`gettype(parse_json(' "abc" '))`	`string`
`gettype(parse_json(' {"abc":1} '))`	`dictionary`

Expression	Returns
`gettype(parse_json(' [1, 2, 3] '))`	`array`
`gettype(123.45)`	`real`
`gettype(guid(12e8b78d-55b4-46ae-b068-26d7a0080254))`	`guid`
`gettype(parse_json(''))`	`null`

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

getyear()

Article • 01/02/2023

Returns the year part of the `datetime` argument.

Syntax

`getyear(`*date*`)`

Parameters

Name	Type	Required	Description
date	datetime	✓	The date for which to get the year.

Returns

The year that contains the given *date*.

Example

Run the query

Kusto

```
print year = getyear(datetime(2015-10-12))
```

year
2015

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

gzip_compress_to_base64_string()

Article • 01/02/2023

Performs gzip compression and encodes the result to base64.

Syntax

`gzip_compress_to_base64_string(`*string*`)`

Parameters

Name	Type	Required	Description
string	string	✓	The value to be compressed and base64 encoded. The function accepts only one argument.

Returns

- Returns a `string` that represents gzip-compressed and base64-encoded original string.
- Returns an empty result if compression or encoding failed.

Example

Run the query

Kusto

```
print res = gzip_compress_to_base64_string("1234567890qwertyuiop")
```

res
H4sIAAAAAAAA/wEUAOv/MTIzNDU2Nzg5MHF3ZXJ0eXVpb3A6m7f2FAAAAA==

See also

- gzip_decompress_from_base64_string()
- zlib_compress_to_base64_string()

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

gzip_decompress_from_base64_string()

Article • 01/02/2023

Decodes the input string from base64 and performs gzip decompression.

Syntax

`gzip_decompress_from_base64_string(`*string*`)`

Parameters

Name	Type	Required	Description
string	string	✓	The value that was compressed with gzip and then base64-encoded. The function accepts only one argument.

> ⓘ Note
>
> - This function checks mandatory gzip header fields (ID1, ID2, and CM) and returns an empty output if any of these fields have incorrect values.
> - The FLG byte is expected to be zero.
> - Optional header fields are not supported.

Returns

- Returns a `string` that represents the original string.
- Returns an empty result if decompression or decoding failed.
  - For example, invalid gzip-compressed and base 64-encoded strings will return an empty output.

Examples

Valid input

Run the query

```
```

Kusto

```
print
res=gzip_decompress_from_base64_string("H4sIAAAAAAAA/wEUAOv/MTIzNDU2Nzg5MHF3
ZXJ0eXVpb3A6m7f2FAAAAA==")
```

res
"1234567890qwertyuiop"

Invalid input

Run the query

Kusto

```
print res=gzip_decompress_from_base64_string("x0x0x0")
```

res

See also

- gzip_compress_to_base64_string()
- zlib_decompress_from_base64_string

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

has_any_index()

Article • 01/30/2023

Searches the string for items specified in the array and returns the position in the array of the first item found in the string. `has_any_index` searches for indexed terms, where an indexed term is three or more characters. If your term is fewer than three characters, the query scans the values in the column, which is slower than looking up the term in the term index.

Syntax

`has_any_index` `(`*source*`,` *values*`)`

Parameters

Name	Type	Required	Description
source	string	✓	The value to search.
values	dynamic	✓	An array of scalar or literal expressions to look up.

Returns

Zero-based index position of the first item in *values* that is found in *source*. Returns -1 if none of the array items were found in the string or if *values* is empty.

Example

Run the query

Kusto

```
print
 idx1 = has_any_index("this is an example", dynamic(['this', 'example']))
// first lookup found in input string
 , idx2 = has_any_index("this is an example", dynamic(['not', 'example']))
// last lookup found in input string
 , idx3 = has_any_index("this is an example", dynamic(['not', 'found'])) //
no lookup found in input string
 , idx4 = has_any_index("Example number 2", range(1, 3, 1)) // Lookup array
of integers
```

```
  , idx5 = has_any_index("this is an example", dynamic([]))  // Empty lookup
array
```

Output

idx1	idx2	idx3	idx4	idx5
0	1	-1	1	-1

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

has_any_ipv4()

Article • 01/02/2023

Returns a value indicating whether one of specified IPv4 addresses appears in a text.

IP address entrances in a text must be properly delimited with non-alphanumeric characters. For example, properly delimited IP addresses are:

- "These requests came from: 192.168.1.1, 10.1.1.115 and 10.1.1.201"
- "05:04:54 127.0.0.1 GET /favicon.ico 404"

Syntax

`has_any_ipv4(` *source* `,` *ip_address* [`,` *ip_address_2*`,` ...] `)`

Parameters

Name	Type	Required	Description
source	string	✓	The value to search.
ip_address	string or dynamic	✓	An IP address, or an array of IP addresses, for which to search.

Returns

`true` if one of specified IP addresses is a valid IPv4 address, and it was found in *source*. Otherwise, the function returns `false`.

Examples

IP addresses as list of strings

Run the query

Kusto

```
print result=has_any_ipv4('05:04:54 127.0.0.1 GET /favicon.ico 404',
'127.0.0.1', '127.0.0.2')
```

result
true

IP addresses as dynamic array

Run the query

Kusto

```
print result=has_any_ipv4('05:04:54 127.0.0.1 GET /favicon.ico 404',
dynamic(['127.0.0.1', '127.0.0.2']))
```

result
true

Invalid IPv4 address

Run the query

Kusto

```
print result=has_any_ipv4('05:04:54 127.0.0.256 GET /favicon.ico 404',
dynamic(["127.0.0.256", "192.168.1.1"]))
```

result
false

Improperly deliminated IP address

Run the query

Kusto

```
print result=has_any_ipv4('05:04:54127.0.0.1 GET /favicon.ico 404',
'127.0.0.1', '192.168.1.1') // false, improperly delimited IP address
```

result

result
false

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

has_ipv4()

Article • 12/28/2022

Returns a value indicating whether a specified IPv4 address appears in a text.

IP address entrances in a text must be properly delimited with non-alphanumeric characters. For example, properly delimited IP addresses are:

- "These requests came from: 192.168.1.1, 10.1.1.115 and 10.1.1.201"
- "05:04:54 127.0.0.1 GET /favicon.ico 404"

Syntax

`has_ipv4(` *source* `,` *ip_address* `)`

Parameters

Name	Type	Required	Description
source	string	✓	The text to search.
ip_address	string	✓	The value containing the IP address for which to search.

Returns

`true` if the *ip_address* is a valid IPv4 address, and it was found in *source*. Otherwise, the function returns `false`.

> Tip
>
> - To search for many IPv4 addresses at once, use **has_any_ipv4()** function.
> - To search for IPv4 addresses prefix, use **has_ipv4_prefix()** function.

Examples

Properly formatted IP address

Run the query

```
Kusto
print result=has_ipv4('05:04:54 127.0.0.1 GET /favicon.ico 404',
'127.0.0.1')
```

Output

result
true

Invalid IP address

Run the query

```
Kusto
print result=has_ipv4('05:04:54 127.0.0.256 GET /favicon.ico 404',
'127.0.0.256')
```

Output

result
false

Improperly delimited IP

Run the query

```
Kusto
print result=has_ipv4('05:04:54127.0.0.1 GET /favicon.ico 404', '127.0.0.1')
```

Output

result
false

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

has_any_ipv4_prefix()

Article • 01/02/2023

Returns a boolean value indicating whether one of specified IPv4 address prefixes appears in a text.

IP address entrances in a text must be properly delimited with non-alphanumeric characters. For example, properly delimited IP addresses are:

- "These requests came from: 192.168.1.1, 10.1.1.115 and 10.1.1.201"
- "05:04:54 127.0.0.1 GET /favicon.ico 404"

Syntax

`has_any_ipv4_prefix(`*source* `,` *ip_address_prefix* [`,` *ip_address_prefix_2*`,` ...] `)`

Parameters

Name	Type	Required	Description
source	string	✓	The value to search.
ip_address_prefix	string or dynamic	✓	An IP address prefix, or an array of IP address prefixes, for which to search. A valid IP address prefix is either a complete IPv4 address, such as `192.168.1.11`, or its prefix ending with a dot, such as `192.`, `192.168.` or `192.168.1.`.

Returns

`true` if the one of specified IP address prefixes is a valid IPv4 address prefix, and it was found in *source*. Otherwise, the function returns `false`.

Examples

IP addresses as list of strings

Run the query

```
```

Kusto

```
print result=has_any_ipv4_prefix('05:04:54 127.0.0.1 GET /favicon.ico 404',
'127.0.', '192.168.') // true
```

result
true

IP addresses as dynamic array

Run the query

Kusto

```
print result=has_any_ipv4_prefix('05:04:54 127.0.0.1 GET /favicon.ico 404',
dynamic(["127.0.", "192.168."]))
```

result
true

Invalid IPv4 prefix

Run the query

Kusto

```
print result=has_any_ipv4_prefix('05:04:54 127.0.0.1 GET /favicon.ico 404',
'127.0')
```

result
false

Improperly deliminated IP address

Run the query

Kusto

```
print result=has_any_ipv4_prefix('05:04:54127.0.0.1 GET /favicon.ico 404',
'127.0.', '192.')
```

result
false

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

has_ipv4_prefix()

Article • 12/29/2022

Returns a value indicating whether a specified IPv4 address prefix appears in a text.

A valid IP address prefix is either a complete IPv4 address (`192.168.1.11`) or its prefix ending with a dot (`192.`, `192.168.` or `192.168.1.`).

IP address entrances in a text must be properly delimited with non-alphanumeric characters. For example, properly delimited IP addresses are:

- "These requests came from: 192.168.1.1, 10.1.1.115 and 10.1.1.201"
- "05:04:54 127.0.0.1 GET /favicon.ico 404"

Syntax

`has_ipv4_prefix(`*source* `,` *ip_address_prefix* `)`

Parameters

Name	Type	Required	Description
source	string	✓	The text to search.
ip_address_prefix	string	✓	The IP address prefix for which to search.

Returns

`true` if the *ip_address_prefix* is a valid IPv4 address prefix, and it was found in *source*. Otherwise, the function returns `false`.

> **Tip**
>
> To search for many IPv4 prefixes at once, use the **has_any_ipv4_prefix()** function.

Examples

Properly formatted IPv4 prefix

Run the query

Kusto

```
print result=has_ipv4_prefix('05:04:54 127.0.0.1 GET /favicon.ico 404',
'127.0.')
```

result
true

Invalid IPv4 prefix

Run the query

Kusto

```
print result=has_ipv4_prefix('05:04:54 127.0.0.1 GET /favicon.ico 404',
'127.0')
```

result
false

Invalid IPv4 address

Run the query

Kusto

```
print result=has_ipv4_prefix('05:04:54 127.0.0.256 GET /favicon.ico 404',
'127.0.')
```

result
false

Improperly delimited IPv4 address

Run the query

Kusto

```
print result=has_ipv4_prefix('05:04:54127.0.0.1 GET /favicon.ico 404',
'127.0.')
```

result
false

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

hash()

Article • 12/29/2022

Returns a hash value for the input value.

> ⓘ Note
>
> - The function calculates hashes using the xxhash64 algorithm, but this may change. It's recommended to only use this function within a single query.
> - If you need to persist a combined hash, it's recommended to use **hash_sha256()**, **hash_sha1()**, or **hash_md5()** and combine the hashes with a **bitwise operator**. These functions are more complex to calculate than `hash()`.

Syntax

`hash(`*source* [`,` *mod*]`)`

Parameters

Name	Type	Required	Description
source	scalar	✓	The value to be hashed.
mod	int		A modulo value to be applied to the hash result, so that the output value is between `0` and `mod - 1`. This parameter is useful for limiting the range of possible output values or for compressing the output of the hash function into a smaller range.

Returns

The hash value of *source*. If *mod* is specified, the function returns the hash value modulo the value of *mod*, meaning that the output of the function will be the remainder of the hash value divided by *mod*. The output will be a value between `0` and `mod - 1`, inclusive.

Examples

String input

Run the query

```
Kusto

print result=hash("World")
```

result
1846988464401551951

String input with mod

Run the query

```
Kusto

print result=hash("World", 100)
```

result
51

Datetime input

Run the query

```
Kusto

print result=hash(datetime("2015-01-01"))
```

result
1380966698541616202

Use hash to check data distribution

Use the `hash()` function for sampling data if the values in one of its columns is uniformly distributed. In the following example, *StartTime* values are uniformly

distributed and the function is used to run a query on 10% of the data.

Run the query

Kusto

```
StormEvents
| where hash(StartTime, 10) == 0
| summarize StormCount = count(), TypeOfStorms = dcount(EventType) by State
| top 5 by StormCount desc
```

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

hash_combine()

Article • 12/29/2022

Combines hash values of two or more hashes.

Syntax

`hash_combine(`*h1* `,` *h2* [`,` *h3* ...]`)`

Parameters

Name	Type	Required	Description
h1, *h2*, ... *hN*	long	✓	The hash values to combine.

Returns

The combined hash value of the given scalars.

Examples

Run the query

Kusto

```
print value1 = "Hello", value2 = "World"
| extend h1 = hash(value1), h2=hash(value2)
| extend combined = hash_combine(h1, h2)
```

Output

value1	value2	h1	h2	combined
Hello	World	753694413698530628	1846988464401551951	-1440138333540407281

Feedback

hash_many()

Article • 12/29/2022

Returns a combined hash value of multiple values.

Syntax

`hash_many(`*s1* `,` *s2* [`,` *s3* ...]`)`

Parameters

Name	Type	Required	Description
s1, *s2*, ..., *sN*	scalar	✓	The values to hash together.

Returns

The hash() function is applied to each of the specified scalars. The resulting hashes are combined into a single hash and returned.

> ⚠ **Warning**
>
> The function uses the *xxhash64* algorithm to calculate the hash for each scalar, but this may change. We therefore only recommend using this function within a single query where all invocations of the function will use the same algorithm.
>
> If you need to persist a combined hash, we recommend using **hash_sha256()**, **hash_sha1()**, or **hash_md5()** and combining the hashes into a single hash with a **bitwise operator**. Note that these functions are more complex to calculate than `hash()`.

Examples

Run the query

Kusto

```
print value1 = "Hello", value2 = "World"
| extend combined = hash_many(value1, value2)
```

Output

value1	value2	combined
Hello	World	-1440138333540407281

Feedback

hash_md5()

Article • 01/26/2023

Returns an MD5 hash value of the input.

Syntax

`hash_md5(`*source*`)`

Parameters

Name	Type	Required	Description
source	scalar	✓	The value to be hashed.

Returns

The MD5 hash value of the given scalar, encoded as a hex string (a string of characters, each two of which represent a single Hex number between 0 and 255).

> ⚠ **Warning**
>
> The algorithm used by this function (MD5) is guaranteed to not be modified in the future, but is very complex to calculate. Users that need a "lightweight" hash function for the duration of a single query are advised to use the function **hash()** instead.

Examples

Run the query

Kusto

```
print
h1=hash_md5("World"),
h2=hash_md5(datetime(2020-01-01))
```

Output

h1	h2
f5a7924e621e84c9280a9a27e1bcb7f6	786c530672d1f8db31fee25ea8a9390b

The following example uses the `hash_md5()` function to aggregate StormEvents based on State's MD5 hash value.

Run the query

Kusto

```
StormEvents
| summarize StormCount = count() by State, StateHash=hash_md5(State)
| top 5 by StormCount
```

Output

State	StateHash	StormCount
TEXAS	3b00dbe6e07e7485a1c12d36c8e9910a	4701
KANSAS	e1338d0ac8be43846cf9ae967bd02e7f	3166
IOWA	6d4a7c02942f093576149db764d4e2d2	2337
ILLINOIS	8c00d9e0b3fcd55aed5657e42cc40cf1	2022
MISSOURI	2d82f0c963c0763012b2539d469e5008	2016

Feedback

hash_sha1()

Article • 02/06/2023

Returns a sha1 hash value of the source input.

Syntax

`hash_sha1(`*source*`)`

Parameters

Name	Type	Required	Description
source	scalar	✓	The value to be hashed.

Returns

The sha1 hash value of the given scalar, encoded as a hex string (a string of characters, each two of which represent a single Hex number between 0 and 255).

> ⚠ Warning
>
> The algorithm used by this function (SHA1) is guaranteed to not be modified in the future, but is very complex to calculate. If you need a "lightweight" hash function for the duration of a single query, consider using hash().

Examples

Run the query

Kusto

```
print
    h1=hash_sha1("World"),
    h2=hash_sha1(datetime(2020-01-01))
```

Output

h1	h2
70c07ec18ef89c5309bbb0937f3a6342411e1fdd	e903e533f4d636b4fc0dcf3cf81e7b7f330de776

The following example uses the `hash_sha1()` function to aggregate StormEvents based on State's SHA1 hash value.

Run the query

Kusto

```
StormEvents
| summarize StormCount = count() by State, StateHash=hash_sha1(State)
| top 5 by StormCount desc
```

Output

State	StateHash	StormCount
TEXAS	3128d805194d4e6141766cc846778eeacb12e3ea	4701
KANSAS	ea926e17098148921e472b1a760cd5a8117e84d6	3166
IOWA	cacf86ec119cfd5b574bde5b59604774de3273db	2337
ILLINOIS	03740763b16dae9d799097f51623fe635d8c4852	2022
MISSOURI	26d938907240121b54d9e039473dacc96e712f61	2016

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

hash_sha256()

Article • 02/06/2023

Returns a sha256 hash value of the source input.

Syntax

`hash_sha256(`*source*`)`

Parameters

Name	Type	Required	Description
source	scalar	✓	The value to be hashed.

Returns

The sha256 hash value of the given scalar, encoded as a hex string (a string of characters, each two of which represent a single Hex number between 0 and 255).

> ⚠ Warning
>
> The algorithm used by this function (SHA256) is guaranteed to not be modified in the future, but is very complex to calculate. Users that need a "lightweight" hash function for the duration of a single query are advised to use the function `hash()` instead.

Examples

Run the query

Kusto

```
print
    h1=hash_sha256("World"),
    h2=hash_sha256(datetime(2020-01-01))
```

Output

h1	h2
78ae647dc5544d227130a0682a51e30bc7777fbb6d8a8f17007463a3ecd1d524	ba666752dc1a20eb750b0eb64e780cc4c968bc9fb8813461c1d7e750f302d71d

The following example uses the `hash_sha256()` function to aggregate StormEvents based on State's SHA256 hash value.

Run the query

Kusto

```
StormEvents
| summarize StormCount = count() by State, StateHash=hash_sha256(State)
| top 5 by StormCount desc
```

Output

State	StateHash	StormCount
TEXAS	9087f20f23f91b5a77e8406846117049029e6798ebbd0d38aea68da73a00ca37	4701
KANSAS	c80e328393541a3181b258cdb4da4d00587c5045e8cf3bb6c8fdb7016b69cc2e	3166

State	StateHash	StormCount
IOWA	f85893dca466f779410f65cd904fdc4622de49e119ad4e7c7e4a291ceed1820b	2337
ILLINOIS	ae3eeabfd7eba3d9a4ccbfed6a9b8cff269dc43255906476282e0184cf81b7fd	2022
MISSOURI	d15dfc28abc3ee73b7d1f664a35980167ca96f6f90e034db2a6525c0b8ba61b1	2016

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

hash_xxhash64()

Article • 12/29/2022

Returns an xxhash64 value for the input value.

Syntax

`hash_xxhash64(`*source* [`,` *mod*]`)`

Parameters

Name	Type	Required	Description
source	scalar	✓	The value to be hashed.
mod	int		A modulo value to be applied to the hash result, so that the output value is between `0` and `mod - 1`. This parameter is useful for limiting the range of possible output values or for compressing the output of the hash function into a smaller range.

Returns

The hash value of *source*. If *mod* is specified, the function returns the hash value modulo the value of *mod*, meaning that the output of the function will be the remainder of the hash value divided by *mod*. The output will be a value between `0` and `mod - 1`, inclusive.

Examples

String input

Run the query

Kusto

```
print result=hash_xxhash64("World")
```

result

result
1846988464401551951

String input with mod

Run the query

```
Kusto
print result=hash_xxhash64("World", 100)
```

result
51

Datetime input

Run the query

```
Kusto
print result=hash_xxhash64(datetime("2015-01-01"))
```

result
1380966698541616202

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

hll_merge()

Article • 06/12/2023

Merges HLL results. This is the scalar version of the aggregate version hll_merge().

Read about the underlying algorithm (*HyperLogLog*) and estimation accuracy.

> ⓘ **Important**
>
> The results of hll(), hll_if(), and hll_merge() can be stored and later retrieved. For example, you may want to create a daily unique users summary, which can then be used to calculate weekly counts. However, the precise binary representation of these results may change over time. There's no guarantee that these functions will produce identical results for identical inputs, and therefore we don't advise relying on them.

Syntax

`hll_merge(` *hll*`,` *hll2*`,` [*hll3*`,` ...]`)`

Parameters

Name	Type	Required	Description
hll, *hll2*, ...	string	✓	The column names containing HLL values to merge. The function expects between 2-64 arguments.

Returns

Returns one HLL value. The value is the result of merging the columns *hll*, *hll2*, ... *hllN*.

Examples

This example shows the value of the merged columns.

Run the query

Kusto

```
range x from 1 to 10 step 1
| extend y = x + 10
| summarize hll_x = hll(x), hll_y = hll(y)
| project merged = hll_merge(hll_x, hll_y)
| project dcount_hll(merged)
```

Output

dcount_hll_merged
20

Estimation accuracy

This function uses a variant of the HyperLogLog (HLL) algorithm, which does a stochastic estimation of set cardinality. The algorithm provides a "knob" that can be used to balance accuracy and execution time per memory size:

Accuracy	Error (%)	Entry count
0	1.6	2^{12}
1	0.8	2^{14}
2	0.4	2^{16}
3	0.28	2^{17}
4	0.2	2^{18}

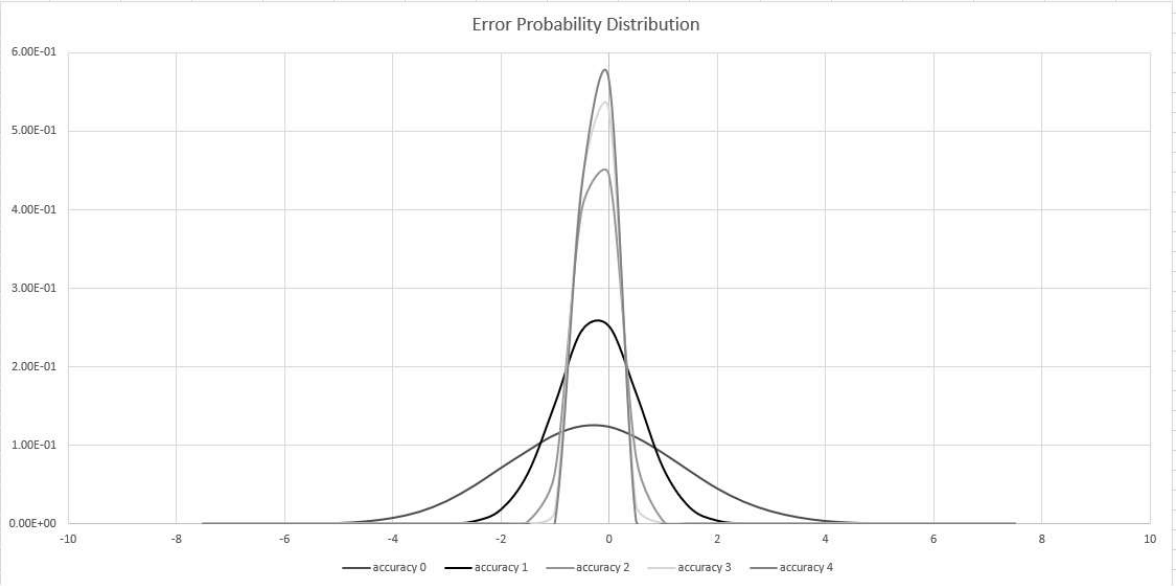
> **Note**
>
> The "entry count" column is the number of 1-byte counters in the HLL implementation.

The algorithm includes some provisions for doing a perfect count (zero error), if the set cardinality is small enough:

- When the accuracy level is `1`, 1000 values are returned
- When the accuracy level is `2`, 8000 values are returned

The error bound is probabilistic, not a theoretical bound. The value is the standard deviation of error distribution (the sigma), and 99.7% of the estimations will have a relative error of under 3 x sigma.

The following image shows the probability distribution function of the relative estimation error, in percentages, for all supported accuracy settings:



Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

hourofday()

Article • 12/29/2022

Returns the integer number representing the hour number of the given date.

Syntax

`hourofday(`*date*`)`

Parameters

Name	Type	Required	Description
date	datetime	✓	The date for which to return the hour number.

Returns

An integer between 0-23 representing the hour number of the day for *date*.

Example

Run the query

Kusto

```
print hour=hourofday(datetime(2015-12-14 18:54))
```

hour
18

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

iff()

Article • 01/16/2023

Returns the value of *then* if *if* evaluates to `true`, or the value of *else* otherwise.

> The `iff()` and `iif()` functions are equivalent

Syntax

`iff(`*if*`,` *then*`,` *else*`)`

Parameters

Name	Type	Required	Description
if	string	✓	An expression that evaluates to a boolean value.
then	scalar	✓	An expression that gets evaluated and its value returned from the function if *if* evaluates to `true`.
else	scalar	✓	An expression that gets evaluated and its value returned from the function if *if* evaluates to `false`.

Returns

This function returns the value of *then* if *if* evaluates to `true`, or the value of *else* otherwise.

Example

Run the query

Kusto

```
StormEvents
| extend Rain = iff((EventType in ("Heavy Rain", "Flash Flood", "Flood")),
"Rain event", "Not rain event")
| project State, EventId, EventType, Rain
```

Output

The following table shows only the first 5 rows.

State	EventId	EventType	Rain
ATLANTIC SOUTH	61032	Waterspout	Not rain event
FLORIDA	60904	Heavy Rain	Rain event
FLORIDA	60913	Tornado	Not rain event
GEORGIA	64588	Thunderstorm Wind	Not rain event
MISSISSIPPI	68796	Thunderstorm Wind	Not rain event
...	...	...	...

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

indexof()

Article • 12/29/2022

Reports the zero-based index of the first occurrence of a specified string within the input string.

For more information, see indexof_regex().

Syntax

`indexof(`*string*`,`*match*`[,`*start*`[,`*length*`[,`*occurrence*`]]])`

Parameters

Name	Type	Required	Description
string	string	✓	The source string to search.
match	string	✓	The string for which to search.
start	int		The search start position. A negative value will offset the starting search position from the end of the *string* by this many steps: `abs(`*start*`)`.
length	int		The number of character positions to examine. A value of -1 means unlimited length.
occurrence	int		The number of the occurrence. The default is 1.

> ⓘ Note
>
> If *string* or *match* isn't of type `string`, the function forcibly casts their value to `string`.

Returns

The zero-based index position of *match*.

- Returns -1 if *match* isn't found in *string*.
- Returns `null` if:
  - *start* is less than 0.

- *occurrence* is less than 0.
- *length* is less than -1.

Examples

Run the query

Kusto

```
print
 idx1 = indexof("abcdefg","cde")    // lookup found in input string
 , idx2 = indexof("abcdefg","cde",1,4) // lookup found in researched range
 , idx3 = indexof("abcdefg","cde",1,2) // search starts from index 1, but
stops after 2 chars, so full lookup can't be found
 , idx4 = indexof("abcdefg","cde",3,4) // search starts after occurrence of
lookup
 , idx5 = indexof("abcdefg","cde",-5)  // negative start index
 , idx6 = indexof(1234567,5,1,4)       // two first parameters were forcibly
casted to strings "12345" and "5"
 , idx7 = indexof("abcdefg","cde",2,-1)  // lookup found in input string
 , idx8 = indexof("abcdefgabcdefg", "cde", 1, 10, 2)   // lookup found in
input range
 , idx9 = indexof("abcdefgabcdefg", "cde", 1, -1, 3)   // the third
occurrence of lookup is not in researched range
```

Output

idx1	idx2	idx3	idx4	idx5	idx6	idx7	idx8	idx9
2	2	-1	-1	2	4	2	9	-1

Feedback

indexof_regex()

Article • 12/29/2022

Returns the zero-based index of the first occurrence of a specified lookup regular expression within the input string.

See indexof().

Syntax

`indexof_regex(`*string*`,`*match*`[,`*start*`[,`*length*`[,`*occurrence*`]]])`

Parameters

Name	Type	Required	Description
string	string	✓	The source string to search.
match	string	✓	The regular expression lookup string.
start	int		The search start position. A negative value will offset the starting search position from the end of the *string* by this many steps: `abs(`*start*`)`.
length	int		The number of character positions to examine. A value of -1 means unlimited length.
occurrence	int		The number of the occurrence. The default is 1.

Returns

The zero-based index position of *match*.

- Returns -1 if *match* isn't found in *string*.
- Returns `null` if:
  - *start* is less than 0.
  - *occurrence* is less than 0.
  - *length* is less than -1.

> ⓘ Note

- Overlapping matches lookup aren't supported.
- Regular expression strings may contain characters that require either escaping or using @'' string-literals.

Examples

Run the query

Kusto

```
print
    idx1 = indexof_regex("abcabc", @"a.c"), // lookup found in input string
    idx2 = indexof_regex("abcabcdefg", @"a.c", 0, 9, 2),  // lookup found in
input string
    idx3 = indexof_regex("abcabc", @"a.c", 1, -1, 2),  // there's no second
occurrence in the search range
    idx4 = indexof_regex("ababaa", @"a.a", 0, -1, 2), // Matches don't
overlap so full lookup can't be found
    idx5 = indexof_regex("abcabc", @"a|ab", -1)  // invalid start argument
```

Output

idx1	idx2	idx3	idx4	idx5
0	3	-1	-1	

Feedback

ingestion_time()

Article • 03/10/2023

Returns the approximate time at which the current record was ingested.

This function must be used in context of a table for which the IngestionTime policy is enabled. Otherwise, this function produces null values.

> **Note**
>
> The value returned by this function is only approximate, as the ingestion process may take several minutes to complete and multiple ingestion activities may take place concurrently. To process all records of a table with exactly-once guarantees, use **database cursors**.

> **Tip**
>
> The ingestion_time() function returns values according to the service clock as measured when ingestion was completed. As a result, this value cannot be used to "order" ingestion operations, as two operations that overlap in time might have any ordering of these values. If ordering records is important for application semantics, one should ensure that the table has a timestamp column as measured by the source of the data instead of relying on the ingestion_time() value.

Syntax

`ingestion_time()`

Returns

A `datetime` value specifying the approximate time of ingestion into a table.

Example

Kusto

```
T
| extend ingestionTime = ingestion_time() | top 10 by ingestionTime
```

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

ipv4_compare()

Article • 02/02/2023

Compares two IPv4 strings. The two IPv4 strings are parsed and compared while accounting for the combined IP-prefix mask calculated from argument prefixes, and the optional `PrefixMask` argument.

Syntax

`ipv4_compare(`*Expr1*`,`*Expr2*`[ ,`*PrefixMask*`])`

Parameters

Name	Type	Required	Description
Expr1, *Expr2*	string	✓	A string expression representing an IPv4 address. IPv4 strings can be masked using IP-prefix notation.
PrefixMask	int		An integer from 0 to 32 representing the number of most-significant bits that are taken into account.

IP-prefix notation

IP-prefix notation (also known as CIDR notation) is a concise way of representing an IP address and its associated network mask. The format is `<base IP>/<prefix length>`, where the prefix length is the number of leading 1 bits in the netmask. The prefix length determines the range of IP addresses that belong to the network.

For IPv4, the prefix length is a number between 0 and 32. So the notation 192.168.2.0/24 represents the IP address 192.168.2.0 with a netmask of 255.255.255.0. This netmask has 24 leading 1 bits, or a prefix length of 24.

For IPv6, the prefix length is a number between 0 and 128. So the notation fe80::85d:e82c:9446:7994/120 represents the IP address fe80::85d:e82c:9446:7994 with a netmask of ffff:ffff:ffff:ffff:ffff:ffff:ffff:ff00. This netmask has 120 leading 1 bits, or a prefix length of 120.

Returns

- `0`: If the long representation of the first IPv4 string argument is equal to the second IPv4 string argument
- `1`: If the long representation of the first IPv4 string argument is greater than the second IPv4 string argument
- `-1`: If the long representation of the first IPv4 string argument is less than the second IPv4 string argument
- `null`: If conversion for one of the two IPv4 strings wasn't successful.

Examples: IPv4 comparison equality cases

Compare IPs using the IP-prefix notation specified inside the IPv4 strings

Run the query

Kusto

```
datatable(ip1_string:string, ip2_string:string)
[
 '192.168.1.0',    '192.168.1.0',       // Equal IPs
 '192.168.1.1/24', '192.168.1.255',     // 24 bit IP-prefix is used for
comparison
 '192.168.1.1',    '192.168.1.255/24',  // 24 bit IP-prefix is used for
comparison
 '192.168.1.1/30', '192.168.1.255/24',  // 24 bit IP-prefix is used for
comparison
]
| extend result = ipv4_compare(ip1_string, ip2_string)
```

Output

ip1_string	ip2_string	result
192.168.1.0	192.168.1.0	0
192.168.1.1/24	192.168.1.255	0
192.168.1.1	192.168.1.255/24	0
192.168.1.1/30	192.168.1.255/24	0

Compare IPs using IP-prefix notation specified inside the IPv4 strings and as additional argument of the

`ipv4_compare()` function

Run the query

Kusto

```
datatable(ip1_string:string, ip2_string:string, prefix:long)
[
 '192.168.1.1',    '192.168.1.0',   31, // 31 bit IP-prefix is used for
comparison
 '192.168.1.1/24', '192.168.1.255', 31, // 24 bit IP-prefix is used for
comparison
 '192.168.1.1',    '192.168.1.255', 24, // 24 bit IP-prefix is used for
comparison
]
| extend result = ipv4_compare(ip1_string, ip2_string, prefix)
```

Output

ip1_string	ip2_string	prefix	result
192.168.1.1	192.168.1.0	31	0
192.168.1.1/24	192.168.1.255	31	0
192.168.1.1	192.168.1.255	24	0

ipv4_is_in_range()

Article • 01/10/2023

Checks if IPv4 string address is in IPv4-prefix notation range.

Syntax

`ipv4_is_in_range(`*Ipv4Address*`,`*Ipv4Range*`)`

Parameters

Name	Type	Required	Description
Ipv4Address	string	✓	An expression representing an IPv4 address.
Ipv4Range	string	✓	An IPv4 range or list of IPv4 ranges written with IP-prefix notation.

IP-prefix notation

IP-prefix notation (also known as CIDR notation) is a concise way of representing an IP address and its associated network mask. The format is `<base IP>/<prefix length>`, where the prefix length is the number of leading 1 bits in the netmask. The prefix length determines the range of IP addresses that belong to the network.

For IPv4, the prefix length is a number between 0 and 32. So the notation 192.168.2.0/24 represents the IP address 192.168.2.0 with a netmask of 255.255.255.0. This netmask has 24 leading 1 bits, or a prefix length of 24.

For IPv6, the prefix length is a number between 0 and 128. So the notation fe80::85d:e82c:9446:7994/120 represents the IP address fe80::85d:e82c:9446:7994 with a netmask of ffff:ffff:ffff:ffff:ffff:ffff:ffff:ff00. This netmask has 120 leading 1 bits, or a prefix length of 120.

Returns

- `true`: If the long representation of the first IPv4 string argument is in range of the second IPv4 string argument.
- `false`: Otherwise.

- `null`: If conversion for one of the two IPv4 strings wasn't successful.

Example

Run the query

Kusto

```
datatable(ip_address:string, ip_range:string)
[
 '192.168.1.1',    '192.168.1.1',       // Equal IPs
 '192.168.1.1',    '192.168.1.255/24',  // 24 bit IP-prefix is used for
comparison
]
| extend result = ipv4_is_in_range(ip_address, ip_range)
```

Output

ip_address	ip_range	result
192.168.1.1	192.168.1.1	true
192.168.1.1	192.168.1.255/24	true

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

ipv4_is_in_any_range()

Article • 01/10/2023

Checks whether IPv4 string address is in any of the specified IPv4 address ranges.

Syntax

`ipv4_is_in_any_range(` *Ipv4Address* `,` *Ipv4Range* [`,` *Ipv4Range* ...] `)`

`ipv4_is_in_any_range(` *Ipv4Address* `,` *Ipv4Ranges* `)`

Parameters

Name	Type	Required	Description
Ipv4Address	string	✓	An expression representing an IPv4 address.
Ipv4Range	string	✓	An IPv4 range or list of IPv4 ranges written with IP-prefix notation.
Ipv4Ranges	dynamic	✓	A dynamic array containing IPv4 ranges written with IP-prefix notation.

> ⓘ Note
>
> Either one or more *IPv4Range* strings or an *IPv4Ranges* dynamic array is required.

IP-prefix notation

IP-prefix notation (also known as CIDR notation) is a concise way of representing an IP address and its associated network mask. The format is `<base IP>/<prefix length>`, where the prefix length is the number of leading 1 bits in the netmask. The prefix length determines the range of IP addresses that belong to the network.

For IPv4, the prefix length is a number between 0 and 32. So the notation 192.168.2.0/24 represents the IP address 192.168.2.0 with a netmask of 255.255.255.0. This netmask has 24 leading 1 bits, or a prefix length of 24.

For IPv6, the prefix length is a number between 0 and 128. So the notation fe80::85d:e82c:9446:7994/120 represents the IP address fe80::85d:e82c:9446:7994 with a

netmask of ffff:ffff:ffff:ffff:ffff:ffff:ffff:ff00. This netmask has 120 leading 1 bits, or a prefix length of 120.

Returns

- `true`: If the IPv4 address is in the range of any of the specified IPv4 networks.
- `false`: Otherwise.
- `null`: If conversion for one of the two IPv4 strings wasn't successful.

Examples

Syntax using list of strings

Run the query

Kusto

```
print Result=ipv4_is_in_any_range('192.168.1.6', '192.168.1.1/24',
'10.0.0.1/8', '127.1.0.1/16')
```

Output

Result
true

Syntax using dynamic array

Run the query

Kusto

```
print Result=ipv4_is_in_any_range("127.0.0.1", dynamic(["127.0.0.1",
"192.168.1.1"]))
```

Output

Result
true

Extend table with IPv4 range check

Run the query

Kusto

```
let LocalNetworks=dynamic([
    "192.168.1.1/16",
    "127.0.0.1/8",
    "10.0.0.1/8"
]);
let IPs=datatable(IP:string) [
    "10.1.2.3",
    "192.168.1.5",
    "123.1.11.21",
    "1.1.1.1"
];
IPs
| extend IsLocal=ipv4_is_in_any_range(IP, LocalNetworks)
```

Output

IP	IsLocal
10.1.2.3	true
192.168.1.5	true
123.1.11.21	false
1.1.1.1	false

Feedback

ipv4_is_match()

Article • 01/10/2023

Matches two IPv4 strings. The two IPv4 strings are parsed and compared while accounting for the combined IP-prefix mask calculated from argument prefixes, and the optional `prefix` argument.

Syntax

`ipv4_is_match(`*ip1*`,`*ip2*`[ ,`*prefix*`])`

Parameters

Name	Type	Required	Description
ip1, *ip2*	string	✓	An expression representing an IPv4 address. IPv4 strings can be masked using IP-prefix notation.
prefix	int		An integer from 0 to 32 representing the number of most-significant bits that are taken into account.

IP-prefix notation

IP-prefix notation (also known as CIDR notation) is a concise way of representing an IP address and its associated network mask. The format is `<base IP>/<prefix length>`, where the prefix length is the number of leading 1 bits in the netmask. The prefix length determines the range of IP addresses that belong to the network.

For IPv4, the prefix length is a number between 0 and 32. So the notation 192.168.2.0/24 represents the IP address 192.168.2.0 with a netmask of 255.255.255.0. This netmask has 24 leading 1 bits, or a prefix length of 24.

For IPv6, the prefix length is a number between 0 and 128. So the notation fe80::85d:e82c:9446:7994/120 represents the IP address fe80::85d:e82c:9446:7994 with a netmask of ffff:ffff:ffff:ffff:ffff:ffff:ffff:ff00. This netmask has 120 leading 1 bits, or a prefix length of 120.

Returns

- `true`: If the long representation of the first IPv4 string argument is equal to the second IPv4 string argument.
- `false`: Otherwise.
- `null`: If conversion for one of the two IPv4 strings wasn't successful.

> ⓘ Note
>
> When matching against an IPv4 address that's not a range, we recommend using the **equals operator** (`==`), for better performance.

Examples

IPv4 comparison equality - IP-prefix notation specified inside the IPv4 strings

Run the query

Kusto

```
datatable(ip1_string:string, ip2_string:string)
[
 '192.168.1.0',    '192.168.1.0',       // Equal IPs
 '192.168.1.1/24', '192.168.1.255',     // 24 bit IP-prefix is used for
comparison
 '192.168.1.1',    '192.168.1.255/24',  // 24 bit IP-prefix is used for
comparison
 '192.168.1.1/30', '192.168.1.255/24',  // 24 bit IP-prefix is used for
comparison
]
| extend result = ipv4_is_match(ip1_string, ip2_string)
```

Output

ip1_string	ip2_string	result
192.168.1.0	192.168.1.0	true
192.168.1.1/24	192.168.1.255	true
192.168.1.1	192.168.1.255/24	true
192.168.1.1/30	192.168.1.255/24	true

IPv4 comparison equality - IP-prefix notation specified inside the IPv4 strings and an additional argument of the `ipv4_is_match()` function

Run the query

Kusto

```
datatable(ip1_string:string, ip2_string:string, prefix:long)
[
 '192.168.1.1',    '192.168.1.0',   31, // 31 bit IP-prefix is used for
comparison
 '192.168.1.1/24', '192.168.1.255', 31, // 24 bit IP-prefix is used for
comparison
 '192.168.1.1',    '192.168.1.255', 24, // 24 bit IP-prefix is used for
comparison
]
| extend result = ipv4_is_match(ip1_string, ip2_string, prefix)
```

Output

ip1_string	ip2_string	prefix	result
192.168.1.1	192.168.1.0	31	true
192.168.1.1/24	192.168.1.255	31	true
192.168.1.1	192.168.1.255	24	true

Feedback

ipv4_is_private()

Article • 01/10/2023

Checks if the IPv4 string address belongs to a set of private network IPs.

Private network addresses were originally defined to assist in delaying IPv4 address exhaustion. IP packets originating from or addressed to a private IP address can't be routed through the public internet.

Private IPv4 addresses

The Internet Engineering Task Force (IETF) has directed the Internet Assigned Numbers Authority (IANA) to reserve the following IPv4 address ranges for private networks:

IP address range	Number of addresses	Largest CIDR block (subnet mask)
10.0.0.0 – 10.255.255.255	16777216	10.0.0.0/8 (255.0.0.0)
172.16.0.0 – 172.31.255.255	1048576	172.16.0.0/12 (255.240.0.0)
192.168.0.0 – 192.168.255.255	65536	192.168.0.0/16 (255.255.0.0)

Kusto

```
ipv4_is_private('192.168.1.1/24') == true
ipv4_is_private('10.1.2.3/24') == true
ipv4_is_private('202.1.2.3') == false
ipv4_is_private("127.0.0.1") == false
```

Syntax

`ipv4_is_private(`*ip*`)`

Parameters

Name	Type	Required	Description
ip	string	✓	An expression representing an IPv4 address. IPv4 strings can be masked using IP-prefix notation.

IP-prefix notation

IP-prefix notation (also known as CIDR notation) is a concise way of representing an IP address and its associated network mask. The format is `<base IP>/<prefix length>`, where the prefix length is the number of leading 1 bits in the netmask. The prefix length determines the range of IP addresses that belong to the network.

For IPv4, the prefix length is a number between 0 and 32. So the notation 192.168.2.0/24 represents the IP address 192.168.2.0 with a netmask of 255.255.255.0. This netmask has 24 leading 1 bits, or a prefix length of 24.

For IPv6, the prefix length is a number between 0 and 128. So the notation fe80::85d:e82c:9446:7994/120 represents the IP address fe80::85d:e82c:9446:7994 with a netmask of ffff:ffff:ffff:ffff:ffff:ffff:ffff:ff00. This netmask has 120 leading 1 bits, or a prefix length of 120.

Returns

- `true`: If the IPv4 address belongs to any of the private network ranges.
- `false`: Otherwise.
- `null`: If parsing of the input as IPv4 address string wasn't successful.

Example: Check if IPv4 belongs to a private network

Run the query

Kusto

```
datatable(ip_string:string)
[
 '10.1.2.3',
 '192.168.1.1/24',
 '127.0.0.1',
]
| extend result = ipv4_is_private(ip_string)
```

Output

ip_string	result
10.1.2.3	true

ip_string	result
192.168.1.1/24	true
127.0.0.1	false

Feedback

ipv4_range_to_cidr_list()

Article • 05/15/2023

Converts a IPv4 address range denoted by starting and ending IPv4 addresses to a list of IPv4 ranges in CIDR notation.

Syntax

`ipv4_range_to_cidr_list(`*StartAddress* `,` *EndAddress* `)`

Parameters

Name	Type	Required	Description
StartAddress	string	✓	An expression representing a starting IPv4 address of the range.
EndAddress	string	✓	An expression representing an ending IPv4 address of the range.

Returns

A dynamic array object containing the list of ranges in CIDR notation.

IP-prefix notation

IP-prefix notation (also known as CIDR notation) is a concise way of representing an IP address and its associated network mask. The format is `<base IP>/<prefix length>`, where the prefix length is the number of leading 1 bits in the netmask. The prefix length determines the range of IP addresses that belong to the network.

For IPv4, the prefix length is a number between 0 and 32. So the notation 192.168.2.0/24 represents the IP address 192.168.2.0 with a netmask of 255.255.255.0. This netmask has 24 leading 1 bits, or a prefix length of 24.

For IPv6, the prefix length is a number between 0 and 128. So the notation fe80::85d:e82c:9446:7994/120 represents the IP address fe80::85d:e82c:9446:7994 with a netmask of ffff:ffff:ffff:ffff:ffff:ffff:ffff:ff00. This netmask has 120 leading 1 bits, or a prefix length of 120.

Examples

Kusto

```
print start_IP="1.1.128.0", end_IP="1.1.140.255"
 | project ipv4_range_list = ipv4_range_to_cidr_list(start_IP, end_IP)
```

Output

ipv4_range_list
`["1.1.128.0/21", "1.1.136.0/22","1.1.140.0/24"]`

Feedback

ipv4_netmask_suffix()

Article • 01/10/2023

Returns the value of the IPv4 netmask suffix from an IPv4 string address.

Syntax

`ipv4_netmask_suffix(`*ip*`)`

Parameters

Name	Type	Required	Description
ip	string	✓	An expression representing an IPv4 address. IPv4 strings can be masked using IP-prefix notation.

IP-prefix notation

IP-prefix notation (also known as CIDR notation) is a concise way of representing an IP address and its associated network mask. The format is `<base IP>/<prefix length>`, where the prefix length is the number of leading 1 bits in the netmask. The prefix length determines the range of IP addresses that belong to the network.

For IPv4, the prefix length is a number between 0 and 32. So the notation 192.168.2.0/24 represents the IP address 192.168.2.0 with a netmask of 255.255.255.0. This netmask has 24 leading 1 bits, or a prefix length of 24.

For IPv6, the prefix length is a number between 0 and 128. So the notation fe80::85d:e82c:9446:7994/120 represents the IP address fe80::85d:e82c:9446:7994 with a netmask of ffff:ffff:ffff:ffff:ffff:ffff:ffff:ff00. This netmask has 120 leading 1 bits, or a prefix length of 120.

Returns

- The value of the netmask suffix the IPv4 address. If the suffix isn't present in the input, a value of `32` (full netmask suffix) is returned.
- `null`: If parsing the input as an IPv4 address string wasn't successful.

Example: Resolve IPv4 mask suffix

Run the query

Kusto

```
datatable(ip_string:string)
[
 '10.1.2.3',
 '192.168.1.1/24',
 '127.0.0.1/16',
]
| extend cidr_suffix = ipv4_netmask_suffix(ip_string)
```

Output

ip_string	cidr_suffix
10.1.2.3	32
192.168.1.1/24	24
127.0.0.1/16	16

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

ipv6_compare()

Article • 01/10/2023

Compares two IPv6 or IPv4 network address strings. The two IPv6 strings are parsed and compared while accounting for the combined IP-prefix mask calculated from argument prefixes, and the optional `prefix` argument.

> ⓘ Note
>
> The function can accept and compare arguments representing both IPv6 and IPv4 network addresses. However, if the caller knows that arguments are in IPv4 format, use **ipv4_is_compare()** function. This function will result in better runtime performance.

Syntax

`ipv6_compare(`*ip1*`,`*ip2*`[ ,`*prefix*`])`

Parameters

Name	Type	Required	Description
ip1, *ip2*	string	✓	An expression representing an IPv6 or IPv4 address. IPv6 and IPv4 strings can be masked using IP-prefix notation.
prefix	int		An integer from 0 to 128 representing the number of most significant bits that are taken into account.

IP-prefix notation

IP-prefix notation (also known as CIDR notation) is a concise way of representing an IP address and its associated network mask. The format is `<base IP>/<prefix length>`, where the prefix length is the number of leading 1 bits in the netmask. The prefix length determines the range of IP addresses that belong to the network.

For IPv4, the prefix length is a number between 0 and 32. So the notation 192.168.2.0/24 represents the IP address 192.168.2.0 with a netmask of 255.255.255.0. This netmask has 24 leading 1 bits, or a prefix length of 24.

For IPv6, the prefix length is a number between 0 and 128. So the notation fe80::85d:e82c:9446:7994/120 represents the IP address fe80::85d:e82c:9446:7994 with a netmask of ffff:ffff:ffff:ffff:ffff:ffff:ffff:ff00. This netmask has 120 leading 1 bits, or a prefix length of 120.

Returns

- `0`: If the long representation of the first IPv6 string argument is equal to the second IPv6 string argument.
- `1`: If the long representation of the first IPv6 string argument is greater than the second IPv6 string argument.
- `-1`: If the long representation of the first IPv6 string argument is less than the second IPv6 string argument.
- `null`: If conversion for one of the two IPv6 strings wasn't successful.

Examples: IPv6/IPv4 comparison equality cases

Compare IPs using the IP-prefix notation specified inside the IPv6/IPv4 strings

Run the query

Kusto

```
datatable(ip1_string:string, ip2_string:string)
[
 // IPv4 are compared as IPv6 addresses
 '192.168.1.1',    '192.168.1.1',       // Equal IPs
 '192.168.1.1/24', '192.168.1.255',     // 24 bit IP4-prefix is used for
comparison
 '192.168.1.1',    '192.168.1.255/24',  // 24 bit IP4-prefix is used for
comparison
 '192.168.1.1/30', '192.168.1.255/24',  // 24 bit IP4-prefix is used for
comparison
  // IPv6 cases
 'fe80::85d:e82c:9446:7994', 'fe80::85d:e82c:9446:7994',         // Equal
IPs
 'fe80::85d:e82c:9446:7994/120', 'fe80::85d:e82c:9446:7998',     // 120 bit
IP6-prefix is used for comparison
 'fe80::85d:e82c:9446:7994', 'fe80::85d:e82c:9446:7998/120',     // 120 bit
IP6-prefix is used for comparison
 'fe80::85d:e82c:9446:7994/120', 'fe80::85d:e82c:9446:7998/120', // 120 bit
IP6-prefix is used for comparison
 // Mixed case of IPv4 and IPv6
```

```
 '192.168.1.1',      '::ffff:c0a8:0101', // Equal IPs
 '192.168.1.1/24',   '::ffff:c0a8:01ff', // 24 bit IP-prefix is used for
comparison
 '::ffff:c0a8:0101', '192.168.1.255/24', // 24 bit IP-prefix is used for
comparison
 '::192.168.1.1/30', '192.168.1.255/24', // 24 bit IP-prefix is used for
comparison
]
| extend result = ipv6_compare(ip1_string, ip2_string)
```

Output

ip1_string	ip2_string	result
192.168.1.1	192.168.1.1	0
192.168.1.1/24	192.168.1.255	0
192.168.1.1	192.168.1.255/24	0
192.168.1.1/30	192.168.1.255/24	0
fe80::85d:e82c:9446:7994	fe80::85d:e82c:9446:7994	0
fe80::85d:e82c:9446:7994/120	fe80::85d:e82c:9446:7998	0
fe80::85d:e82c:9446:7994	fe80::85d:e82c:9446:7998/120	0
fe80::85d:e82c:9446:7994/120	fe80::85d:e82c:9446:7998/120	0
192.168.1.1	::ffff:c0a8:0101	0
192.168.1.1/24	::ffff:c0a8:01ff	0
::ffff:c0a8:0101	192.168.1.255/24	0
::192.168.1.1/30	192.168.1.255/24	0

Compare IPs using IP-prefix notation specified inside the IPv6/IPv4 strings and as additional argument of the `ipv6_compare()` function

Run the query

Kusto

```
datatable(ip1_string:string, ip2_string:string, prefix:long)
[
 // IPv4 are compared as IPv6 addresses
```

```
 '192.168.1.1',    '192.168.1.0',   31, // 31 bit IP4-prefix is used for
comparison
 '192.168.1.1/24', '192.168.1.255', 31, // 24 bit IP4-prefix is used for
comparison
 '192.168.1.1',    '192.168.1.255', 24, // 24 bit IP4-prefix is used for
comparison
   // IPv6 cases
 'fe80::85d:e82c:9446:7994', 'fe80::85d:e82c:9446:7995',     127, // 127 bit
IP6-prefix is used for comparison
 'fe80::85d:e82c:9446:7994/127', 'fe80::85d:e82c:9446:7998', 120, // 120 bit
IP6-prefix is used for comparison
 'fe80::85d:e82c:9446:7994/120', 'fe80::85d:e82c:9446:7998', 127, // 120 bit
IP6-prefix is used for comparison
 // Mixed case of IPv4 and IPv6
 '192.168.1.1/24',   '::ffff:c0a8:01ff', 127, // 127 bit IP6-prefix is used
for comparison
 '::ffff:c0a8:0101', '192.168.1.255',    120, // 120 bit IP6-prefix is used
for comparison
 '::192.168.1.1/30', '192.168.1.255/24', 127, // 120 bit IP6-prefix is used
for comparison
]
| extend result = ipv6_compare(ip1_string, ip2_string, prefix)
```

Output

ip1_string	ip2_string	prefix	result
192.168.1.1	192.168.1.0	31	0
192.168.1.1/24	192.168.1.255	31	0
192.168.1.1	192.168.1.255	24	0
fe80::85d:e82c:9446:7994	fe80::85d:e82c:9446:7995	127	0
fe80::85d:e82c:9446:7994/127	fe80::85d:e82c:9446:7998	120	0
fe80::85d:e82c:9446:7994/120	fe80::85d:e82c:9446:7998	127	0
192.168.1.1/24	::ffff:c0a8:01ff	127	0
::ffff:c0a8:0101	192.168.1.255	120	0
::192.168.1.1/30	192.168.1.255/24	127	0

Feedback

ipv6_is_in_range()

Article • 01/10/2023

Checks if an IPv6 string address is in the IPv6-prefix notation range.

Syntax

`ipv6_is_in_range(`*Ipv6Address*`,`*Ipv6Range*`)`

Parameters

Name	Type	Required	Description
Ipv6Address	string	✓	An expression representing an IPv6 address.
Ipv6Range	string	✓	An expression representing an IPv6 range using IP-prefix notation.

IP-prefix notation

IP-prefix notation (also known as CIDR notation) is a concise way of representing an IP address and its associated network mask. The format is `<base IP>/<prefix length>`, where the prefix length is the number of leading 1 bits in the netmask. The prefix length determines the range of IP addresses that belong to the network.

For IPv4, the prefix length is a number between 0 and 32. So the notation 192.168.2.0/24 represents the IP address 192.168.2.0 with a netmask of 255.255.255.0. This netmask has 24 leading 1 bits, or a prefix length of 24.

For IPv6, the prefix length is a number between 0 and 128. So the notation fe80::85d:e82c:9446:7994/120 represents the IP address fe80::85d:e82c:9446:7994 with a netmask of ffff:ffff:ffff:ffff:ffff:ffff:ffff:ff00. This netmask has 120 leading 1 bits, or a prefix length of 120.

Returns

- `true`: If the long representation of the first IPv6 string argument is in range of the second IPv6 string argument.
- `false`: Otherwise.

- `null`: If conversion for one of the two IPv6 strings wasn't successful.

Example

Run the query

Kusto

```
datatable(ip_address:string, ip_range:string)
[
 'a5e:f127:8a9d:146d:e102:b5d3:c755:abcd',
'a5e:f127:8a9d:146d:e102:b5d3:c755:0000/112',
 'a5e:f127:8a9d:146d:e102:b5d3:c755:abcd',
'a5e:f127:8a9d:146d:e102:b5d3:c755:abcd',
 'a5e:f127:8a9d:146d:e102:b5d3:c755:abcd',    '0:0:0:0:0:ffff:c0a8:ac/60',
]
| extend result = ipv6_is_in_range(ip_address, ip_range)
```

Output

ip_address	ip_range	result
a5e:f127:8a9d:146d:e102:b5d3:c755:abcd	a5e:f127:8a9d:146d:e102:b5d3:c755:0000/112	True
a5e:f127:8a9d:146d:e102:b5d3:c755:abcd	a5e:f127:8a9d:146d:e102:b5d3:c755:abcd	True
a5e:f127:8a9d:146d:e102:b5d3:c755:abcd	0:0:0:0:0:ffff:c0a8:ac/60	False

Feedback

ipv6_is_in_any_range()

Article • 01/10/2023

Checks whether an IPv6 string address is in any of the specified IPv6 address ranges.

Syntax

`ipv6_is_in_any_range(` *Ipv6Address* `,` *Ipv6Range* [`,` *Ipv6Range* ...] `)`

`ipv6_is_in_any_range(` *Ipv6Address* `,` *Ipv6Ranges* `)`

Parameters

Name	Type	Required	Description
Ipv6Address	string	✓	An expression representing an IPv6 address.
Ipv6Range	string	✓	An expression representing an IPv6 range using IP-prefix notation.
Ipv6Ranges	dynamic	✓	An array containing IPv6 ranges using IP-prefix notation.

> ⓘ Note
>
> Either one or more *IPv6Range* strings or an *IPv6Ranges* dynamic array is required.

IP-prefix notation

IP-prefix notation (also known as CIDR notation) is a concise way of representing an IP address and its associated network mask. The format is `<base IP>/<prefix length>`, where the prefix length is the number of leading 1 bits in the netmask. The prefix length determines the range of IP addresses that belong to the network.

For IPv4, the prefix length is a number between 0 and 32. So the notation 192.168.2.0/24 represents the IP address 192.168.2.0 with a netmask of 255.255.255.0. This netmask has 24 leading 1 bits, or a prefix length of 24.

For IPv6, the prefix length is a number between 0 and 128. So the notation fe80::85d:e82c:9446:7994/120 represents the IP address fe80::85d:e82c:9446:7994 with a

netmask of ffff:ffff:ffff:ffff:ffff:ffff:ffff:ff00. This netmask has 120 leading 1 bits, or a prefix length of 120.

Returns

- `true`: If the IPv6 address is in the range of any of the specified IPv6 networks.
- `false`: Otherwise.
- `null`: If conversion for one of the two IPv6 strings wasn't successful.

Example

Run the query

Kusto

```
let LocalNetworks=dynamic([
    "a5e:f127:8a9d:146d:e102:b5d3:c755:f6cd/112",
    "0:0:0:0:0:ffff:c0a8:ac/60"
]);
let IPs=datatable(IP:string) [
    "a5e:f127:8a9d:146d:e102:b5d3:c755:abcd",
    "a5e:f127:8a9d:146d:e102:b5d3:c755:abce",
    "a5e:f127:8a9d:146d:e102:b5d3:c755:abcf",
    "a5e:f127:8a9d:146d:e102:b5d3:c756:abd1",
];
IPs
| extend IsLocal=ipv6_is_in_any_range(IP, LocalNetworks)
```

Output

IP	IsLocal
a5e:f127:8a9d:146d:e102:b5d3:c755:abcd	True
a5e:f127:8a9d:146d:e102:b5d3:c755:abce	True
a5e:f127:8a9d:146d:e102:b5d3:c755:abcf	True
a5e:f127:8a9d:146d:e102:b5d3:c756:abd1	False

Feedback

ipv6_is_match()

Article • 01/10/2023

Matches two IPv6 or IPv4 network address strings. The two IPv6/IPv4 strings are parsed and compared while accounting for the combined IP-prefix mask calculated from argument prefixes, and the optional `prefix` argument.

> ⓘ Note
>
> The function can accept and compare arguments representing both IPv6 and IPv4 network addresses. If the caller knows that arguments are in IPv4 format, use the **ipv4_is_match()** function. This function will result in better runtime performance.

Syntax

`ipv6_is_match(`*ip1*`,`*ip2*`[ ,`*prefix*`])`

Parameters

Name	Type	Required	Description
ip1, *ip2*	string	✓	An expression representing an IPv6 or IPv4 address. IPv6 and IPv4 strings can be masked using IP-prefix notation.
prefix	int		An integer from 0 to 128 representing the number of most-significant bits that are taken into account.

IP-prefix notation

IP-prefix notation (also known as CIDR notation) is a concise way of representing an IP address and its associated network mask. The format is `<base IP>/<prefix length>`, where the prefix length is the number of leading 1 bits in the netmask. The prefix length determines the range of IP addresses that belong to the network.

For IPv4, the prefix length is a number between 0 and 32. So the notation 192.168.2.0/24 represents the IP address 192.168.2.0 with a netmask of 255.255.255.0. This netmask has 24 leading 1 bits, or a prefix length of 24.

For IPv6, the prefix length is a number between 0 and 128. So the notation fe80::85d:e82c:9446:7994/120 represents the IP address fe80::85d:e82c:9446:7994 with a netmask of ffff:ffff:ffff:ffff:ffff:ffff:ffff:ff00. This netmask has 120 leading 1 bits, or a prefix length of 120.

Returns

- `true`: If the long representation of the first IPv6/IPv4 string argument is equal to the second IPv6/IPv4 string argument.
- `false`: Otherwise.
- `null`: If conversion for one of the two IPv6/IPv4 strings wasn't successful.

Examples

IPv6/IPv4 comparison equality case - IP-prefix notation specified inside the IPv6/IPv4 strings

Run the query

Kusto

```
datatable(ip1_string:string, ip2_string:string)
[
 // IPv4 are compared as IPv6 addresses
 '192.168.1.1',    '192.168.1.1',       // Equal IPs
 '192.168.1.1/24', '192.168.1.255',     // 24 bit IP4-prefix is used for
comparison
 '192.168.1.1',    '192.168.1.255/24',  // 24 bit IP4-prefix is used for
comparison
 '192.168.1.1/30', '192.168.1.255/24',  // 24 bit IP4-prefix is used for
comparison
  // IPv6 cases
 'fe80::85d:e82c:9446:7994', 'fe80::85d:e82c:9446:7994',         // Equal
IPs
 'fe80::85d:e82c:9446:7994/120', 'fe80::85d:e82c:9446:7998',     // 120 bit
IP6-prefix is used for comparison
 'fe80::85d:e82c:9446:7994', 'fe80::85d:e82c:9446:7998/120',     // 120 bit
IP6-prefix is used for comparison
 'fe80::85d:e82c:9446:7994/120', 'fe80::85d:e82c:9446:7998/120', // 120 bit
IP6-prefix is used for comparison
 // Mixed case of IPv4 and IPv6
 '192.168.1.1',      '::ffff:c0a8:0101', // Equal IPs
 '192.168.1.1/24',   '::ffff:c0a8:01ff', // 24 bit IP-prefix is used for
comparison
 '::ffff:c0a8:0101', '192.168.1.255/24', // 24 bit IP-prefix is used for
```

```
comparison
 '::192.168.1.1/30', '192.168.1.255/24', // 24 bit IP-prefix is used for
comparison
]
| extend result = ipv6_is_match(ip1_string, ip2_string)
```

Output

ip1_string	ip2_string	result
192.168.1.1	192.168.1.1	1
192.168.1.1/24	192.168.1.255	1
192.168.1.1	192.168.1.255/24	1
192.168.1.1/30	192.168.1.255/24	1
fe80::85d:e82c:9446:7994	fe80::85d:e82c:9446:7994	1
fe80::85d:e82c:9446:7994/120	fe80::85d:e82c:9446:7998	1
fe80::85d:e82c:9446:7994	fe80::85d:e82c:9446:7998/120	1
fe80::85d:e82c:9446:7994/120	fe80::85d:e82c:9446:7998/120	1
192.168.1.1	::ffff:c0a8:0101	1
192.168.1.1/24	::ffff:c0a8:01ff	1
::ffff:c0a8:0101	192.168.1.255/24	1
::192.168.1.1/30	192.168.1.255/24	1

IPv6/IPv4 comparison equality case- IP-prefix notation specified inside the IPv6/IPv4 strings and as additional argument of the `ipv6_is_match()` function

Run the query

Kusto

```
datatable(ip1_string:string, ip2_string:string, prefix:long)
[
 // IPv4 are compared as IPv6 addresses
 '192.168.1.1',    '192.168.1.0',   31, // 31 bit IP4-prefix is used for
comparison
 '192.168.1.1/24', '192.168.1.255', 31, // 24 bit IP4-prefix is used for
comparison
```

```
 '192.168.1.1',    '192.168.1.255', 24, // 24 bit IP4-prefix is used for
comparison
   // IPv6 cases
 'fe80::85d:e82c:9446:7994', 'fe80::85d:e82c:9446:7995',     127, // 127 bit
IP6-prefix is used for comparison
 'fe80::85d:e82c:9446:7994/127', 'fe80::85d:e82c:9446:7998', 120, // 120 bit
IP6-prefix is used for comparison
 'fe80::85d:e82c:9446:7994/120', 'fe80::85d:e82c:9446:7998', 127, // 120 bit
IP6-prefix is used for comparison
 // Mixed case of IPv4 and IPv6
 '192.168.1.1/24',   '::ffff:c0a8:01ff', 127, // 127 bit IP6-prefix is used
for comparison
 '::ffff:c0a8:0101', '192.168.1.255',    120, // 120 bit IP6-prefix is used
for comparison
 '::192.168.1.1/30', '192.168.1.255/24', 127, // 120 bit IP6-prefix is used
for comparison
]
| extend result = ipv6_is_match(ip1_string, ip2_string, prefix)
```

Output

ip1_string	ip2_string	prefix	result
192.168.1.1	192.168.1.0	31	1
192.168.1.1/24	192.168.1.255	31	1
192.168.1.1	192.168.1.255	24	1
fe80::85d:e82c:9446:7994	fe80::85d:e82c:9446:7995	127	1
fe80::85d:e82c:9446:7994/127	fe80::85d:e82c:9446:7998	120	1
fe80::85d:e82c:9446:7994/120	fe80::85d:e82c:9446:7998	127	1
192.168.1.1/24	::ffff:c0a8:01ff	127	1
::ffff:c0a8:0101	192.168.1.255	120	1
::192.168.1.1/30	192.168.1.255/24	127	1

Feedback

isascii()

Article • 01/03/2023

Returns `true` if the argument is a valid ASCII string.

Syntax

`isascii(`*value*`)`

Parameters

Name	Type	Required	Description
value	string	✓	The value to check if a valid ASCII string.

Returns

A boolean value indicating whether *value* is a valid ASCII string.

Example

Run the query

Kusto

```
print result=isascii("some string")
```

Output

result
true

Feedback

Was this page helpful? Yes No

isempty()

Article • 01/03/2023

Returns `true` if the argument is an empty string or is null.

Syntax

`isempty(`*value*`)`

Parameters

Name	Type	Required	Description
value	string	✓	The value to check if empty or null.

Returns

A boolean value indicating whether *value* is an empty string or is null.

Example

x	isempty(x)
""	true
"x"	false
parsejson("")	true
parsejson("[]")	false
parsejson("{}")	false

Feedback

isfinite()

Article • 01/03/2023

Returns whether the input is a finite value, meaning it's neither infinite nor NaN.

Syntax

`isfinite(`*number*`)`

Parameters

Name	Type	Required	Description
number	real	✓	The value to check if finite.

Returns

`true` if x is finite and `false` otherwise.

Example

Run the query

Kusto

```
range x from -1 to 1 step 1
| extend y = 0.0
| extend div = 1.0*x/y
| extend isfinite=isfinite(div)
```

Output

x	y	div	isfinite
-1	0	-∞	0
0	0	NaN	0
1	0	∞	0

See also

- To check if a value is null, see isnull().
- To check if a value is infinite, see isinf().
- To check if a value is NaN (Not-a-Number), see isnan().

Feedback

isinf()

Article • 01/03/2023

Returns whether the input is an infinite (positive or negative) value.

Syntax

`isinf(`*number*`)`

Parameters

Name	Type	Required	Description
number	real	✓	The value to check if infinite.

Returns

`true` if x is a positive or negative infinite and `false` otherwise.

Example

Run the query

Kusto

```
range x from -1 to 1 step 1
| extend y = 0.0
| extend div = 1.0*x/y
| extend isinf=isinf(div)
```

Output

x	y	div	isinf
-1	0	-∞	true
0	0	NaN	false
1	0	∞	true

See also

- To check if a value is null, see isnull().
- To check if a value is finite, see isfinite().
- To check if a value is NaN (Not-a-Number), see isnan().

Feedback

isnan()

Article • 01/03/2023

Returns whether the input is a Not-a-Number (NaN) value.

Syntax

`isnan(`*number*`)`

Parameters

Name	Type	Required	Description
number	scalar	✓	The value to check if NaN.

Returns

`true` if x is NaN and `false` otherwise.

Example

Run the query

Kusto

```
range x from -1 to 1 step 1
| extend y = (-1*x)
| extend div = 1.0*x/y
| extend isnan=isnan(div)
```

Output

x	y	div	isnan
-1	1	-1	false
0	0	NaN	true
1	-1	-1	false

See also

- To check if a value is null, see isnull().
- To check if a value is finite, see isfinite().
- To check if a value is infinite, see isinf().

Feedback

isnotempty()

Article • 01/03/2023

Returns `true` if the argument isn't an empty string, and it isn't null.

> **Deprecated aliases:** notempty()

Syntax

`isnotempty(`*value*`)`

Parameters

Name	Type	Required	Description
value	scalar	✓	The value to check if not empty or null.

Returns

`true` if *value* is not null and `false` otherwise.

Example

Find the storm events for which there's a begin location.

Run the query

Kusto

```
StormEvents
| where isnotempty(BeginLat) and isnotempty(BeginLon)
```

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

isnotnull()

Article • 01/03/2023

Returns `true` if the argument isn't null.

> **Deprecated aliases:** notnull()

Syntax

`isnotnull(`*value*`)`

Parameters

Name	Type	Required	Description
value	scalar	✓	The value to check if not null.

Returns

`true` if *value* is not null and `false` otherwise.

Example

Find the storm events for which there's a begin location.

Run the query

Kusto

```
StormEvents
| where isnotnull(BeginLat) and isnotnull(BeginLon)
```

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

isnull()

Article • 01/03/2023

Evaluates its sole argument and returns a `bool` value indicating if the argument evaluates to a null value.

> ⓘ Note
>
> String values can't be null. Use **isempty** to determine if a value of type `string` is empty or not.

Syntax

`isnull(`*Expr*`)`

Parameters

Name	Type	Required	Description
value	scalar	✓	The value to check if not null.

Returns

`true` if *value* is not null and `false` otherwise.

x	`isnull(x)`
`""`	`false`
`"x"`	`false`
`parse_json("")`	`true`
`parse_json("[]")`	`false`
`parse_json("{}")`	`false`

Example

Find the storm events for which there's not a begin location.

Run the query

Kusto

```
StormEvents
| where isnull(BeginLat) and isnull(BeginLon)
```

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

isutf8()

Article • 01/03/2023

Returns `true` if the argument is a valid UTF8 string.

Syntax

`isutf8(`*value*`)`

Parameters

Name	Type	Required	Description
value	string	✓	The value to check if a valid UTF8 string.

Returns

A boolean value indicating whether *value* is a valid UTF8 string.

Example

Run the query

Kusto

```
print result=isutf8("some string")
```

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

jaccard_index()

Article • 01/03/2023

Calculates the Jaccard index of two input sets.

Syntax

`jaccard_index`(*set1*, *set2*)

Parameters

Name	Type	Required	Description
set1	dynamic	✓	The array representing the first set for the calculation.
set2	dynamic	✓	The array representing the second set for the calculation.

> ⓘ Note
>
> Duplicate values in the input arrays are ignored.

Returns

The Jaccard index of the two input sets. The Jaccard index formula is |*set1* ∩ *set2*| / |*set1* ∪ *set2*|.

Examples

Run the query

Kusto

```
print set1=dynamic([1,2,3]), set2=dynamic([1,2,3,4])
| extend jaccard=jaccard_index(set1, set2)
```

Output

set1	set2	jaccard

set1	set2	jaccard
[1,2,3]	[1,2,3,4]	0.75

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

log()

Article • 01/10/2023

The natural logarithm is the base-e logarithm: the inverse of the natural exponential function (exp).

Syntax

`log(`*number*`)`

Parameters

Name	Type	Required	Description
number	real	✓	The number for which to calculate the logarithm.

Returns

- `log()` returns the natural logarithm of the input.
- `null` if the argument is negative or null or can't be converted to a `real` value.

Example

Run the query

Kusto

```
print result=log(5)
```

Output

result
1.6094379124341003

See also

- For common (base-10) logarithms, see log10().

- For base-2 logarithms, see log2().

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

log10()

Article • 01/10/2023

`log10()` returns the common (base-10) logarithm of the input.

Syntax

`log10(`*number*`)`

Parameters

Name	Type	Required	Description
number	real	✓	The number for which to calculate the base-10 logarithm.

Returns

- The common logarithm is the base-10 logarithm: the inverse of the exponential function (exp) with base 10.
- `null` if the argument is negative or null or can't be converted to a `real` value.

Example

Run the query

Kusto

```
print result=log10(5)
```

Output

result
0.69897000433601886

See also

- For natural (base-e) logarithms, see log().

- For base-2 logarithms, see log2()

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

log2()

Article • 01/10/2023

The logarithm is the base-2 logarithm: the inverse of the exponential function (exp) with base 2.

Syntax

`log2(`*number*`)`

Parameters

Name	Type	Required	Description
number	real	✓	The number for which to calculate the base-2 logarithm.

Returns

- The logarithm is the base-2 logarithm: the inverse of the exponential function (exp) with base 2.
- `null` if the argument is negative or null or can't be converted to a `real` value.

Example

Run the query

Kusto

```
print result=log2(5)
```

Output

result
2.3219280948873622

See also

- For natural (base-e) logarithms, see log().
- For common (base-10) logarithms, see log10().

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

loggamma()

Article • 01/10/2023

Computes log of the absolute value of the gamma function

Syntax

`loggamma(`*number*`)`

Parameters

Name	Type	Required	Description
number	real	✓	The number for which to calculate the gamma.

Example

Run the query

Kusto

```
print result=loggamma(5)
```

Output

result
3.1780538303479458

Returns

- Returns the natural logarithm of the absolute value of the gamma function of x.
- For computing gamma function, see gamma().

Feedback

Was this page helpful? Yes No