02_Atoms_basic_data_types

September 22, 2019

1 2. Basic data types: atoms

Atomic data types in q and their counterparts in SQL Java

- Namespaces of data types in q that can be used for casting boolean? byte? short int: ‘int$1976.01.12

long real float char symbol timestamp month date: `date$123`

(datetime) timespan minute second time enumeration table dictionary function nil item

1.1 2.1 Integer Data

1.1.1 2.1.1 long

- long: the basic data type is a 64-bit, long, signed integer (range: -2^63 - 2^63-1)

1.1.2 2.1.2 short and int

[2]:
```
/ type promotion: narrower types are not promoted automatically
type_prom:(4h;5h;6h;2h)
tp2:3e,type_prom
tp3:3h,type_prom
tp2
```

[2]:
```
3e
4h
5h
6h
2h
```

[3]:
```
tp3
```

[3]:
```
3 4 5 6 2h
```

1.2 2.2 Floating Point Data

1.2.1 2.2.1 float

- float: 64 bit signed floating point number (same as double in Java) (conforming to IEEE specification) forward slash is for comments therefore division is written as %!!!!!!!!! ### 2.2.2 real ### 2.2.3 Floating Point Display

[ ]:
```
\P 0 / sets the floating point display to maximum
sqrt 2
```

1.3 2.3 Binary Data

2.3.1 boolean - boolean values - true: 1b - false: 0b

[ ]:
```
0b / false
1b / true
```

[ ]:
```
 / boolean values are converted to integers in arithmetic expression: 0b to 0␣
↪and 1b to 1
1+1b
1+0b
2.3*0b
```

[ ]:
```
 / Tip: The ability of booleans to participate in arithmetic can be useful in␣
↪eliminating conditionals.
flag:1b
base:100
base+flag*42
```

2.3.2 byte - byte stores 1 byte = 8 bits - denoted by 0x plus the hexadecimal digits

[ ]:
```
0x00
```

2.3.3 GUID - Globally Unique IDentifier - 0Ng is the null guid: its value is (00000000-0000-0000-0000-000000000000)

[ ]:
```
3?0Ng / ? as deal operator generates a list of guids, with 3 elements in this␣
↪case
```

TIP: The difference between using a positive integer vs. a negative integer to generate a list of GUIDs is that the positive case uses the same initial seed in each new q session whereas the negative case uses a random seed. The former is useful for reproducible results during testing but only the latter should be used in production; otherwise, your "GUIDs" will not be unique across q sessions.

[ ]:
```
-3?0Ng
```

You can import a guid generated elsewhere by parsing a string of 16 hex digits. - use ”G”$”uuid in string format” to parse a string uuid into q’s guid You can also convert from a list of 16 bytes using an overload of sv - convert a 16-byte byte list to guid

[ ]:
```
parsed_guid:"G"$"61f35174-90bc-a48a-d88f-e15e4a377ec8"
parsed_guid
16?0xff
0x0 sv 16?0xff
```

The only operations available for guids are ~, =, <, > and null.

1.4 2.4 Text Data

2.4.1 char - `char` - 8 bit long - one character in double quotes 2.4.2 symbol - ”symbol” is an atomic data type for storing textual data: same as VARCHAR in SQL and string in other languages - `backtick` + an arbitrary number of characters: ‘iamasymbol - Symbols are used for names in q. All names are symbols but not all symbols are names. - symbols are irreducable: individual characters are not accessible - a symbol is not a string (in q, a list of chars is a string)

[ ]:
```
`a~"a" / a symbol is not a string
```

[ ]:
```
 / casting arbitrary character sequences to symbol: `$"characters to cast"
`$"A symbol with blanks and ` |\\,*(){}!&@" / backslash needs to be escaped
```

1.5 2.5 Temporal Data

- q has a nanosecond-based temporal data type system (metric prefixes)
- q extends the basic sql date and time data types
- you can carry out temporal operations (arithmetic calculations) on temporal types

2.5.1 date - `date` is the number of days since the millennium (2000.01.01 00:00:00), positive for post and negative for pre - notation: yyyy.mm.dd - The underlying value is the count of days from Jan 1, 2000 – positive for post-millennium and negative for pre

[ ]:
```
`int$2000.02.01 / cast date to int to get the underlying value
`date$(`int$1953.12.29 - `int$1976.01.12) / cast integer value to date
```

2.5.2 Time Types - `time`: measures time in milliseconds from midnight - notation: hh:mm:ss.uuu - `timespan`: measures time in nanoseconds from midnight - It is a long integer count of the number of nanoseconds (10 xexp -9) since midnight - notation: hh:mm:ss.nnnnnnnnn

[ ]:
```
`int$00:00:00.001 / time stored in milliseconds
`int$00:00:00.000000001 / time stored in nanoseconds
`int$00:00:00.00000001 / time stored in nanoseconds, trailing zeros do not have
 ↪to be written
```

2.5.3 Date-Time Types - there are two types: - `datetime`: DEPRICATED!!!! - `timestamp`: the lexical combination of date and timespan - Post-millennium is positive and pre- is negative - timestamp format: dateDtime 1976.01.12D13:30:00.00000000

[4]:
```
`long$1976.01.12D13:30:00.00000000 / the underlying nanosecond count can be
↪obtained by casting to long
```

[4]: -756383400000000000

[5]:
```
 / extracting date and time from timestamp by casting to date and time
`date$1976.01.12
`timespan$13:30:00.00000000
```

[5]: 1976.01.12

[5]: 0D13:30:00.000000000

2.5.4 month - counts the number of months from the millenium - yyyy.mm+m - example: 2019.02m

[ ]:
```
2019.02m
2019.01 / !!!!! Leaving off the type indicator m yields a float. This is a
↪common qbie error.
`int$2015.01m / month count

2015.07m=2015.07.01 / this is true. why? or how?
2015.07m=2015.07.02 / this is false
```

2.5.5 minute - number of minutes from midnight - 32-bit integer - notation: hh:mm

[ ]:
```
 / A minute equals its equivalent time and timestamp counterparts
12:00=12:00:00.000
12:00=12:00:00.000000000
```

2.5.6 second - The second type is stored as 32-bit signed integer - notation: hh:mm:ss - A second value counts the number of seconds from midnight.

[ ]:
```
23:59:59
23:59:59=-1+24*60*60
24*60*60
24*60*60-1
24*60*59
-1+24*60*60
```

[ ]:
```
`int$12:34:56
`int$12:34:56.000
`long$12:34:56.000000000
```

[ ]:
```
/
these values are equal in the eyes of q - as they should be,
since they are merely representations in different units of the same position␣
↪on a clock.
\
12:34:56=12:34:56.000
12:34:56.000=12:34:56.000000000
```

2.5.7 Constituents and Dot Notation - The constituents of compound temporal types can be extracted using dot notation. - Unfortunately, at the time of this writing (Sep 2015) dot notation for extraction (still) does not work inside functions. - So cast instead - For example, the field values of a date are named year, mm and dd; similarly for time and other temporal types:

[ ]:
```
dt:2014.01.01
dt.year
dt.mm
dt.dd
ti:12:34:56.789
ti.hh
ti.mm
ti.ss
`year$dt / year as q data type
`month$dt / month as q data type
`mm$dt / months
`dd$dt / days
`int$dt / minutes
(`int$12:34:56.789) mod 1000 / casting milliseconds
(`long$12:34:56.123456789) mod 1000000000 / nanoseconds
```

[ ]:
```
myTimes + `timestamp$myDates / merge date and timespan into timestamp
```

1.6 2.6 Arithmetic Infinities and Nulls

- 0w: positive float infinity
- -0w: Negative float infinity
- 0n: Null float ; NaN, or not a number
- 0W: Positive long infinity
- -0W: Negative long infinity
- 0N: Null long
- In q, division of numeric values always results in a float!!!
- In mathematics, division of a positive value by 0 results in positive infinity and
- division of a negative value by zero results in negative infinity.

[6]:
```
1 % 0
error:-1 % 0 / dividing by 0 produces a special float value rather than
 ↪throwing an exception
0 % 0
0w % 0w
-0w % 0w
0 % 0w
0w % 0
1+error
```

[6]: 0w

[6]: 0n

[6]: 0n

[6]: 0n

[6]: 0f

[6]: 0w

[6]: -0w

[7]:
```
 / The integral infinities do produce the correct results in comparisons; in
 ↪fact, this is their raison d'être:
42<0W
-0W<42
```

[7]: 1b

[7]: 1b

- q does not trap overflow: so adding numbers to the positive integer infinity results in numbers counting from negative integer infinity upwards.
- Implementing proper arithmetic on integer infinities would entail expensive tests in the arithmetic operators and an unacceptable slow-down for normal arithmetic.

[8]:
```
0W+1
0W+2
0W+3
```

[8]: 0N

[8]: -0W

[8]: -9223372036854775806

1.7 2.7 Nulls

2.7.0 Overview of Nulls - The concept of a null value generally indicates missing data. This is an area in which q differs from both traditional programming languages and SQL. - In q, there are no pointers or references: so null cannot mean an unallocated entity - Some types do not designate a distinct null value because there is no available bit pattern: - i.e., for boolean, byte and char all underlying bit patterns are meaningfully employed. - In this case, the **value with no information content** serves as a proxy for null. - Not all data types have null values - only those have a null value where it is meaningfule to distinguish missing data from data whose underlying value is zero - An advantage of the q approach is that the null values act like other values in expressions. The tradeoff is that you must use the correct null value in type-checked situations. - Null values in different data types: boolean* 0b guid* 0Ng (00000000-0000-0000-0000-000000000000) byte* 0x00 short 0Nh Int 0N long 0Nj real 0Ne float 0n char* " " - The value "" is not a null char. It is an empty list of char. sym `ignore this:` timestamp 0Np month 0Nm date 0Nd datetime 0Nz timespan 0Nn minute 0Nu second 0Nv time 0Nt

2.7.1 Binary Nulls - has no null values 2.7.2 Numeric and Temporal Nulls 2.7.3 Text Nulls

2.7.4 Testing for Null - NOT ADVISED: You could test for null using = but this requires a null literal of correct type. Because q is dynamically typed, this can result in problems if a variable changes type during program execution. - ADVISED: Always use the monadic `null` to test a value for null, as opposed to =, as it provides a type-independent check. Also, you don't have to remember the funky null literals. - Avoids null pointer exception, a billion dollar mistake

[9]:
```
null 42
null `
null " "
null ""
```

[9]: 0b

[9]: 1b

[9]: 1b

[9]: `boolean$()