Parsing data in kdb+

Passing data between systems can often become complex, if not impossible, due to lack of interoperability. While kdb+ provides many interfaces on https://code.kx.com/q/interfaces/ to simplify integration its native text parsing capabilities remain extremely important too as they can be used to greatly simplify data ingestion and inter-process communication. These requirements are often encountered exploration or proof-of-concept phases of projects but there are two additional areas where they can be critical:

- For large once off ingestion of data
- For ongoing ingestion of data from a provider which does not provide other methods.

In both of these cases it is important the parsing of data is as efficient as possible. In this blog we offer some tips to implement clean and efficient parsing of CSV formats to kdb+ and in a follow-up blog we will look at the increasingly popular alternative format, JSON.

CSV

CSV files do not have flexibility in the type of data structures they hold but parsing is straight forward.

Take the following example file:

```
read0 `:tab.csv
```

```
"longCol,floatCol,symbolCol,stringCol,skipCol,dateCol,timeCol,ignoreCol"
"1,4,b,bb,8,2018-11-23,00:01:00.000,--"
"2,5,h,dd,8,2018-11-23,00:01:00.003,--"
```

With one line it was possible to parse to the exact type of table required

```
tab:("JFS* DT";enlist ",") 0: `:tab.csv
tab
meta tab
```

```
longCol floatCol symbolCol stringCol dateCol    timeCol
-------------------------------------------------------------
1       4        b         "bb"      2018.11.23 00:01:00.000
2       5        h         "dd"      2018.11.23 00:01:00.003

c        | t f a
---------| -----
longCol  | j
floatCol | f
```

```
symbolCol| s
stringCol| C
dateCol  | d
timeCol  | t
```

- `*` is used to read a string column. `stringCol`
- A space can be used to ignore a given column. `skipCol`
- Any column to the right which is not given a parse rule is ignored. `ignoreCol`

For more information, setting and the loading of fixed-width fields see:

- https://code.kx.com/q/ref/filenumbers/#load-csv

Don't clog your memory

A common mistake by users loading CSV data is to use `S` as a default datatype. This is the symbol datatype which should only be used when loading columns that have a small number of unique values. Instead `*` when uncertain, this will load the data as strings (`C` in a table meta).

This is because symbols are interned strings. Any time a symbol of a new value is created in a process it will be added to an internal lookup. These cannot be freed during garbage collection with .Q.gc and will instead persist in the memory of the process until it exits.

You can see the number of syms in this table and the space memory they are using with .Q.w

```
.Q.w[]`syms`symw //Check the start values
```

```
1206 53636
```

```
r:`$string til 10000 //Create a list of 10 thousand symbols
r
.Q.w[]`syms`symw
```

```
`0`1`2`3`4`5`6`7`8`9`10`11`12`13`14`15`16`17`18`19`20`21`22`23`24`25`26`27`28..

11204 588234
```

```
delete r from `. //Delete r
.Q.gc[] //Run garbage collection
.Q.w[]`syms`symw //No memory is returned to the process
```

```
`.

0

1200 53438
```

The white paper Working With Sym Files covers this topic in greater detail when it comes time to store symbols to disk.

Complex parsing

Date format

- Use \z to control parsing of dates. Set to 0 for "mm/dd/yyyy" and 1 for "dd/mm/yyyy".
    - https://code.kx.com/q/ref/syscmds/#z-date-parsing

```
//You may get an unexpected date
\z 0
"D"$"06/01/2010"
```

```
2010.06.01
```

```
//Or a null
\z 0
"D"$"30/12/2010"
```

```
0Nd
```

```
//Changing z to 1 gives the intended result
\z 1
"D"$"06/01/2010"
```

```
2010.01.06
```

```
//Using \z correctly will perform much better than using a manual parsing method
manyDates:100000#enlist "30/12/2010"
```

```
\t "D"${"." sv reverse "/" vs x} each manyDates
```

```
98
```

```
\z 1
\t "D"$manyDates
```

```
7
```

Other accepted time and date formats

Many other formats can be parsed by kdb+.

A selection:

```
"D"$"2018-12-30"
```

```
2018.12.30
```

```
"D"$"2018 Jan 30"
```

```
2018.01.30
```

```
"D"$"2018 January 30"
```

```
2018.01.30
```

```
"P"$"1546300800" //Unix epoch
```

```
2019.01.01D00:00:00.000000000
```

```
//These value only parse to the deprecated datetime Z format
//We can simply cast them to timestamps
`timestamp$"Z"$"2019-01-01T00:00:00.000Z"
```

```
2019.01.01D00:00:00.000000000
```

Speed and efficiency

Don't do the same work twice (.Q.fu)

There are often fields which we cannot parse natively. Parsing using custom string manipulations is a computationally intensive task. One way to avoid this is by applying the function once per distinct item and mapping the result. This is only suitable when the data has a smaller number of distinct elements in it. .i.e for dates but not unique timestamps etc.

`.Q.fu` simplifies this task

```
//Take this example which will not parse:
"D"$"November 30 2018"
```

```
0Nd
```

```
//By reordering the components it will parse
"D"$" " sv @[;2 0 1] " " vs "November 30 2018"
```

```
2018.11.30
```

```
//This text based cutting is not efficient
manyDates:100000#enlist "November 30 2018"
\t "D"${" " sv @[;2 0 1] " " vs x} each manyDates
```

```
166
```

```
//Switching to .Q.fu sees a huge speed up
\t .Q.fu[{"D"${" " sv @[;2 0 1] " " vs x} each x}] manyDates
```

```
17
```

Straight line speed (Vectorised operations)

Sometimes part of parsing data requires mathematical calculations be performed. A common example of this is differing epochs between languages and systems. When parsing a column one may write functions which iterate through a row at a time rather than operating on the whole column. This is sometimes the only choice. However if suitable kdb+ has native vector based operations which gain huge efficiency by operating on the column as a whole.

```
//Assume we are given a field which is seconds since 1900.01.01D00
//"3755289600"
//With that information we can extract what is needed from the field
1900.01.01D00+0D00:00:01*"J"$"3755289600"
```

```
2019.01.01D00:00:00.000000000
```

```
//If may be tempting to write a function and iterate through the data
manyTimes:1000000#enlist "3755289600"
\t {1900.01.01D00+0D00:00:01*"J"$x} each manyTimes
//But this will perform poorly
```

```
593
```

```
//It serves better to write functions which accept lists
//This allows you to take advantage of vector based numeric operators in cases
like this
\t {1900.01.01D00+0D00:00:01*"J"$x} manyTimes
```

```
34
```

Skip the middle man (Named pipes)

Often plain text files will come compressed. This requires them to be:

1. Decompressed to disk in full
2. Ingested from disk

This is an inefficient use of resources, as the uncompressed file will only ever be read once. Named pipes allow the disk to be taken out of the equation by streaming the uncompressed data directly to kdb+

For more information and examples see: https://code.kx.com/q/cookbook/named-pipes/

Stream it in (.Q.fs, .Q.fsn & .Q.fps)

As text files grow the memory usage of the ingestion process can become a concern. `.Q.fs`,`.Q.fsn` and `.Q.fps` allow control of this by providing the ability to specify the number of lines at a time to pull in to memory. Then each batch can be published to another process on written to disk before continuing.

- `.Q.fs` - Operates on standard files
- `.Q.fsn` - Allows the specification of chunk size
- `.Q.fps` - Operates on named pipes

As well as memory management `.Q.fsn` also allows us to ensure our optimizations using `.Q.fu` and vectorised operations are supplied with sufficient data on each invocation to see speed ups.

- https://code.kx.com/q/ref/dotq/#qfs-streaming-algorithm
- https://code.kx.com/q/ref/dotq/#qfsn-streaming-algorithm
- https://code.kx.com/q/ref/dotq/#qfps-streaming-algorithm