Parsing data in kdb+

Passing data between systems can often become complex if not impossible due to lack of interoperability. Kdb+ has many available interfaces but it's native text parsing skills remain extremely important.

Using text based formats simplifies communication. CSV traditionally and now often JSON are used for this purpose. These can sometimes be used only during exploration phases before more efficient and direct methods are set in place.

However this text based basing of data can also be part of:

1. A large once off ingestion of data.
2. An ongoing ingestion of data from a provider which does not provide other methods.

In both of these cases we want to ensure the parsing of data is as efficient as possible.

This notebook offers some tips to implement clean and efficient parsing of textual formats to kdb+.

CSV

CSV files do not have flexibility in the type 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`

It is important to use `*` as your default rule rather than `S`. This is to ensure the process memory does not become bloated with unnecessary interned strings.

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

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

Complex parsing

Date format

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

```
\z 0
"D"$"06/01/2010"
```

```
2010.06.01
```

```
\z 1
"D"$"06/01/2010"
```

```
2010.01.06
```

This is much preferred than the alternative

```
\z 0
"D"$"30/12/2010"
```

```
0Nd
```

```
manyDates:100000#enlist "30/12/2010"
\t "D"${"." sv reverse "/" vs x} each manyDates
```

```
99
```

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

```
7
```

Other accepted formats

Many other formats can be parsed by kdb+

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

```
2018.12.30
```

```
"D"$"2018 Jan 30"
//ToDo: flesh out this list with all accepted
```

```
2018.01.30
```

Speed and efficiency

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

There are commonly fields which we cannot parse natively. Completing these custom string manipulations can be a computationally intensive task. One way to avoid this is by applying the function once per distinct item and mapping the result. For this reason it is only of benefit when the data has a smaller number of distinct elements in it.

.i.e suitable for dates but not unique timestamps etc.

`.Q.fu` is the inbuilt function which provides 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
```

```
130
```

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

```
12
```

Straight line speed (Vectorised operations)

Sometimes as part of parsing data mathematical calculations will need to be performed. A common example of this is differing epochs between languages and systems.

When parsing a column one may write functions which iterate though a field at a time rather than operating on the whole column. This is sometimes the only choice. However if calculations are involved 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
```

```
447
```

```
//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
```

```
32
```

Skip the middle man (named pipes)

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

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

This step in 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.fps)

As text files grow the memory usage of the ingestion process can become a concern. `.Q.fs` 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.fps` - Operates on named pipes

As well as memory management `.Q.fps` 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/#qfps-streaming-algorithm

JSON

JSON can hold more complex structures than CSV files which is useful but can cause added complexity during ingestion.

Basic datatype support also means we cannot simply rely on the inbuilt `.j` functions in kdb+

```
//Roundtrip fails
6~.j.k .j.j 6
```

```
0b
```

```
//All numerics are devolved to floats
.j.k .j.j 6
```

```
6f
```

- https://code.kx.com/q/ref/dotj/

JSON table encoding

```
//Create a sample table
tab:([] longCol:1 2;
        floatCol:4 5f;
        symbolCol:`b`h;
        stringCol:("bb";"dd");
        dateCol:2018.11.23 2018.11.23;
        timeCol:00:01:00.000 00:01:00.003)
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
```

```
meta tab
```

```
c        | t f a
---------| -----
longCol  | j
floatCol | f
symbolCol| s
stringCol| C
dateCol  | d
timeCol  | t
```

```
//Round trip to JSON results in many differences
.j.k .j.j tab
meta .j.k .j.j 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  | f
floatCol | f
symbolCol| C
stringCol| C
dateCol  | C
timeCol  | C
```

```
//Use lower case casts on numerics and captial case tok on string type data
//* will leave a column untouched
flip "jfS*DT"$flip .j.k .j.j tab
tab~flip "jfS*DT"$flip .j.k .j.j 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

1b
```

- https://code.kx.com/q/ref/casting/#cast
- https://code.kx.com/q/ref/casting/#tok

Field based JSON encoding

JSON can hold more complex data structures than csv

One common example are dictionaries

```
\c 25 200
read0 `:sample.json
```

```
"
{\"data\":\"26cd02c57f9db87b1df9f2e7bb20cc7b2c47e077ff5b0aa0866568bf5036bb65\",\"e
xpiry\":1527796725,\"requestID\":[\"b4a566eb-2529-5cf4-1327-857e3d73653e\"]}"
"{\"result\":\"success\",\"message\":\"success\",\"receipt\":
[123154,4646646],\"requestID\":[\"b4a566eb-2529-5cf4-1327-857e3d73653e\"]}"
"{\"receipt\":[12345678,98751466],\"requestID\":[\"b4a566eb-2529-5cf4-1327-
857e3d73653e\"]}"
"
{\"data\":\"26cd02c57f9db87b1df9f2e7bb20cc7b2c47e077ff5b0aa0866568bf5036bb65\",\"r
equestID\":[\"b4a566eb-2529-5cf4-1327-857e3d73653e\"]}"
"{\"receipt\":[12345678,98751466],\"requestID\":[\"b4a566eb-2529-5cf4-1327-
857e3d73653e\"]}"
"
{\"listSize\":2,\"list\":\"lzplogjxokyetaeflilquziatzpjagsginnajfpbkomfancdmhmumxh
azblddhccprlxtreageghmwtmyeyabavpbcksadnirgddymljslffrplcerdvhbvvshhmcpev\"}"
"{\"requestID\":[\"b4a566eb-2529-5cf4-1327-857e3d73653e\"]}"
```

One way to manage these items may be to create a utility which will cast any dictionary using `key` values to control casting rules.

This can allow more complex parsing rules for each field.

```
//Converts JSON to q with rules per key
decode:{[j]k:.j.k j;(key k)!j2k[key k]@'value k}
```

```
//Converts q to JSON with rules per key
encode:{[k].j.j (key k)!k2j[key k]@'value k}

//Rules for JSON to q conversion
j2k:(enlist `)!enlist (::);

j2k[`expiry]:{0D00:00:01*`long$x};
j2k[`result]:`$;
j2k[`receipt]:`long$;
j2k[`id]:{"G"$first x};
j2k[`listSize]:`long$;
j2k[`data]:cut[64];
j2k[`blockCount]:`long$;
j2k[`blocks]:raze;

//Rules for q to JSON conversion
k2j:(enlist `)!enlist (::);

k2j[`expiry]:{`long$%[x;0D00:00:01]};
k2j[`result]:(::);
k2j[`receipt]:(::);
k2j[`id]:enlist;
k2j[`listSize]:(::);
k2j[`data]:raze;
k2j[`blocks]:(::);
```

```
//Using default .j.k our structures are not transferred as we wish
{show .j.k x} each read0 `:sample.json;
```

```
data     | "26cd02c57f9db87b1df9f2e7bb20cc7b2c47e077ff5b0aa0866568bf5036bb65"
expiry   | 1.527797e+009
requestID| ,"b4a566eb-2529-5cf4-1327-857e3d73653e"
result   | "success"
message  | "success"
receipt  | 123154 4646646f
requestID| ,"b4a566eb-2529-5cf4-1327-857e3d73653e"
receipt  | 1.234568e+007 9.875147e+007
requestID| ,"b4a566eb-2529-5cf4-1327-857e3d73653e"
data     | "26cd02c57f9db87b1df9f2e7bb20cc7b2c47e077ff5b0aa0866568bf5036bb65"
requestID| ,"b4a566eb-2529-5cf4-1327-857e3d73653e"
receipt  | 1.234568e+007 9.875147e+007
requestID| ,"b4a566eb-2529-5cf4-1327-857e3d73653e"
listSize| 2f
list     |
"lzplogjxokyetaeflilquziatzpjagsginnajfpbkomfancdmhmumxhazblddhccprlxtreageghmwtmy
eyabavpbcksadnirgddymljslffrplcerdvhbvvshhmcpev"
requestID| "b4a566eb-2529-5cf4-1327-857e3d73653e"
```

```
//Using decode utility captures complex structures
{show decode x} each read0 `:sample.json;
```

```
data    | ,"26cd02c57f9db87b1df9f2e7bb20cc7b2c47e077ff5b0aa0866568bf5036bb65"
expiry   | 17682D19:58:45.000000000
requestID| ,"b4a566eb-2529-5cf4-1327-857e3d73653e"
result   | `success
message  | "success"
receipt  | 123154 4646646
requestID| ,"b4a566eb-2529-5cf4-1327-857e3d73653e"
receipt  | 12345678 98751466
requestID| ,"b4a566eb-2529-5cf4-1327-857e3d73653e"
data     | "26cd02c57f9db87b1df9f2e7bb20cc7b2c47e077ff5b0aa0866568bf5036bb65"
requestID| "b4a566eb-2529-5cf4-1327-857e3d73653e"
receipt  | 12345678 98751466
requestID| ,"b4a566eb-2529-5cf4-1327-857e3d73653e"
listSize| 2
list    |
"lzplogjxokyetaeflilquziatzpjagsginnajfpbkomfancdmhmumxhazblddhccprlxtreageghmwtmy
eyabavpbcksadnirgddymljslffrplcerdvhbvvshhmcpev"
requestID| "b4a566eb-2529-5cf4-1327-857e3d73653e"
```

```
//The encode utility allows us to roundtrip
{sample~{encode decode x} each sample:read0 x}`:sample.json
```

```
1b
```

Querying unstructured data

With the release of Anymap in kdb+ 3.6 unstructured data has become much easier to manage in kdb+.

However, some considerations do need to be taken in to account.

- https://code.kx.com/q/ref/releases/ChangesIn3.6/#anymap

```
sample:([] data:decode each read0 `:sample.json)
sample
```

```
data
---------------------------------------------------------------------------------
```

```
---------------------------------------------------------------------------
`data`expiry`requestID!
(,"26cd02c57f9db87b1df9f2e7bb20cc7b2c47e077ff5b0aa0866568bf5036bb65";17682D19:58:4
5.000000000;,"b4a566eb-2529-5cf4-1327-857e3d73653e")
`result`message`receipt`requestID!(`success;"success";123154 4646646;,"b4a566eb-
2529-5cf4-1327-857e3d73653e")
`receipt`requestID!(12345678 98751466;,"b4a566eb-2529-5cf4-1327-857e3d73653e")
`data`requestID!
(,"26cd02c57f9db87b1df9f2e7bb20cc7b2c47e077ff5b0aa0866568bf5036bb65";,"b4a566eb-
2529-5cf4-1327-857e3d73653e")
`receipt`requestID!(12345678 98751466;,"b4a566eb-2529-5cf4-1327-857e3d73653e")
`listSize`list!
(2;"lzplogjxokyetaeflilquziatzpjagsginnajfpbkomfancdmhmumxhazblddhccprlxtreageghmw
tmyeyabavpbcksadnirgddymljslffrplcerdvhbvvshhmcpev")
(,`requestID)!,,"b4a566eb-2529-5cf4-1327-857e3d73653e"
```

Indexing at depth allows the sparse data within the dictionaries to be queried easily

```
select data[;`requestID] from sample
```

```
x
---------------------------------------
,"b4a566eb-2529-5cf4-1327-857e3d73653e"
,"b4a566eb-2529-5cf4-1327-857e3d73653e"
,"b4a566eb-2529-5cf4-1327-857e3d73653e"
,"b4a566eb-2529-5cf4-1327-857e3d73653e"
,"b4a566eb-2529-5cf4-1327-857e3d73653e"
0N
,"b4a566eb-2529-5cf4-1327-857e3d73653e"
```

When a key is missing from a dictionary kdb+ will return a `null` value.

The type of this null is determined by the type of the first key within the dictionary.

This poses an issue.

```
//Many different nulls are returned
select data[;`expiry] from sample
```

```
x
------------------------
17682D19:58:45.000000000
`

`long$()
```

```
,""
`long$()
0N
,""
```

```
//Succeds on first 2 rows as by chance only null returned in a atom null
select from (2#sample) where null data[;`expiry]
//Fails once moving to 3 rows as there is an empty list null
select from (3#sample) where null data[;`expiry]
```

```
data
-----------------------------------------------------------------------------------
---------------------------
`result`message`receipt`requestID!(`success;"success";123154 4646646;,"b4a566eb-
2529-5cf4-1327-857e3d73653e")

evaluation error:
type
  [0]  select from (3#sample) where null data[;`expiry]
       ^
```

Checking if a given key in in the rows dictionary will only return rows which do not have the key

```
select from sample where `expiry in/:key each data, not null data[;`expiry]
```

```
data
-----------------------------------------------------------------------------------
---------------------------------------------------------------------------
`data`expiry`requestID!
(,"26cd02c57f9db87b1df9f2e7bb20cc7b2c47e077ff5b0aa0866568bf5036bb65";17682D19:58:4
5.000000000;,"b4a566eb-2529-5cf4-1327-857e3d73653e")
```

If we prepend each dictionary with the null symbol key ``` and generic null value `(::)` we now can query in a more free manner.

```
update data:(enlist[`]!enlist (::))(,)/:data from `sample;
sample
```

```
data
--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
``data`expiry`requestID!
(::;,"26cd02c57f9db87b1df9f2e7bb20cc7b2c47e077ff5b0aa0866568bf5036bb65";17682D19:5
8:45.000000000;,"b4a566eb-2529-5cf4-1327-857e3d73653e")
``result`message`receipt`requestID!(::;`success;"success";123154
4646646;,"b4a566eb-2529-5cf4-1327-857e3d73653e")
``receipt`requestID!(::;12345678 98751466;,"b4a566eb-2529-5cf4-1327-857e3d73653e")
``data`requestID!
(::;,"26cd02c57f9db87b1df9f2e7bb20cc7b2c47e077ff5b0aa0866568bf5036bb65";,"b4a566eb
-2529-5cf4-1327-857e3d73653e")
``receipt`requestID!(::;12345678 98751466;,"b4a566eb-2529-5cf4-1327-857e3d73653e")
``listSize`list!
(::;2;"lzplogjxokyetaeflilquziatzpjagsginnajfpbkomfancdmhmumxhazblddhccprlxtreageg
hmwtmyeyabavpbcksadnirgddymljslffrplcerdvhbvvshhmcpev")
``requestID!(::;,"b4a566eb-2529-5cf4-1327-857e3d73653e")
```

All nulls when a given key is missing are now `(::)`

```
select expiry:data[;`expiry] from sample
```

```
expiry
------------------------
17682D19:58:45.000000000
::
::
::
::
::
::
```

The previously failing query can now execute as there are no list type nulls

```
select from sample where not null data[;`expiry]
```

```
data
--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
``data`expiry`requestID!
(::;,"26cd02c57f9db87b1df9f2e7bb20cc7b2c47e077ff5b0aa0866568bf5036bb65";17682D19:5
8:45.000000000;,"b4a566eb-2529-5cf4-1327-857e3d73653e")
```

These `(::)` can also be replaced with filled with chosen values.

Here an infinite value is chosen:

```
fill:{@[y;where null y;:;x]}
select expiry:fill[0Wn]data[;`expiry] from sample
```

```
expiry
------------------------
17682D19:58:45.000000000
0W
0W
0W
0W
0W
0W
```