it's about time

Technical Whitepaper

Parse Trees and Functional Forms

Author:

Peter Storeng is a kdb+ developer who has worked as a consultant for some of the world's largest financial institutions. He is currently based in London where he divides his time between maintaining a firm wide risk system and implementing algorithmic trading strategies for a tier 1 investment bank.

TABLE OF CONTENTS

1 INTRODUCTION

The importance of being able to properly understand and use the functional form of qSql statements in kdb+ cannot be overstated. The functional form has many advantages over the traditional qSql approach, including the ability to dynamically select columns and build where clauses. It is important for any q programmer to fully understand the functional form and how to convert from qSql to this form correctly.

Applying `parse` to a qSql statement written as a string will return the internal representation of the functional form. With some manipulation this can then be used to piece together the functional form in q. This generally becomes more difficult as the query becomes more complex and requires a deep understanding of what kdb+ is doing when it parses qSql form.

The main goal of this paper is to show in detail how this conversion works, so that it is understood how to correctly build the functional form of qSql statements. In order to do this, we will need to look at how q and k commands relate to each other, as the parse function often returns the cryptic k code for functions. An understanding of what parse trees are, and how to use them, is also vital in the building of functional queries.

Finally, this paper will look at creating a function which will automate the process of converting qSql statements into functional form. This is to be used as a helpful development tool when facing the task of writing a tricky functional statement.

All tests were run using kdb+ version 3.2 (2015.01.14).

2 k4, q AND q.k

kdb+ is a database management system which comes with the general purpose and database language q. q is a less terse syntactic variant of the 4th revision of k, k4. In fact the interpreter can switch between q and k4 modes and run both scripts written in q and k4.

There is no real need to write code in k as all the same functionality can be achieved in the much more readable q language. However in certain cases such as debugging, it can be useful to have at least a basic understanding.

The q.k file comes as part of the standard installation of q and is loaded into each q session on startup. It contains the definitions of the keywords in the q language in terms of k4. To see how a q keyword is defined in terms of k we could check the q.k file or simply enter it into the q prompt:

```
q)key
!:
```

Note that a few are defined natively from C and so do not have a k representation:

```
q)like
like
```

3 THE PARSE KEYWORD

parse is a useful tool for seeing how a statement in q is evaluated. Pass the parse command a q statement as a string and it will return the parse tree of that expression.

3.1 Parse Trees

A parse tree is a q construct which represents an expression but which is not immediately evaluated. It takes the form of a list where the first element is a function and the remaining entries are the arguments. Any of the elements of the list can be parse trees themselves.

Note that in a parse tree a variable is represented by a symbol containing its name.
Thus to distinguish a symbol or a list of symbols from a variable it is necessary to enlist that expression. When we apply the parse command to create a parse tree explicit definitions in .q are shown in their full k form. In particular an enlisted element is represented by a preceding comma.

```
q)parse"5 6 7 8 + 1 2 3 4"
+                   //the function/operator
5 6 7 8             //first argument
1 2 3 4             //second argument

q)parse"2+4*7"
+                   //the function/operator
2                   //first argument
(*;4;7)             //second argument, itself a parse tree

q)v:`e`f
q)`a`b`c,`d,v
`a`b`c`d`e`f
q)parse"`a`b`c,`d,v"
,                   // join operator
,`a`b`c             //actual symbols/lists of symbols are enlisted
(,;,`d;`v)          //v a variable represented as a symbol
```

We can also manually construct a parse tree:

```
q)show pTree:parse "(aggr;data) fby grp"
k){@[(#y)#x[0]0#x 1;g;:;x[0]'x[1]g:.=y]}  //fby in k form
(enlist;`aggr;`data)
`grp
q)
q)
q)pTree~(fby;(enlist;`aggr;`data);`grp)  //manually constructed
1b                                       //parse tree
```

As asserted previously every statement in q parses into the form:

`(function;arg 1;……;arg n)` where every element could itself be a parse tree. In this way we see that every action in q is essentially a function evaluation.

3.2 eval & value

`eval` can be thought of as the dual to `parse`. The following holds for all valid q statements put into a string (recall that `value` executes the command inside a string)

```
q)value[str]~eval parse str
1b              //a tautology (for all valid q expressions str)

q)value["2+4*7"]~eval parse"2+4*7"  //simple example
1b
```

When passed a list, `value` applies the first element (which contains a function) to the rest of the list (the arguments):

```
q) function[arg 1;..;arg n] ~ value(function;arg 1;..;arg n)
1b
```

When `eval` and `value` operate on a parse tree with no nested parse trees they return the same result. However it is not true that `eval` and `value` are equivalent in general. `eval` operates on parse trees, evaluating any nested parse trees, whereas `value` operates on the literals:

```
q)value(+;7;3)            //parse tree, with no nested trees
10
q)eval(+;7;3)
10
q)eval(+;7;(+;2;1))       //parse tree with nested trees
10
q)value(+;7;(+;2;1))
'type
```

```
q)value(,;`a;`b)
`a`b
q)eval(,;`a;`b)           //no variable b defined
'b
q)eval(,;enlist `a;enlist `b)
`a`b
```

3.3 Operator Ambivalence

Many of the built in kdb+ operators in both k and q are overloaded so that the behaviour of the operator changes depending on the number and type of arguments. In q, but not k, the monadic functionality of special character operators (+,$,.,& etc.) is disabled and names are provided instead.

For example, in k the monadic $ operator equates to the string operator in q.

```
q)$42                  //$ monadic functionality disabled in q
'$
q)k)$42
"42"
q)string 42
"42"
```

When a special character operator is wrapped in brackets and not combined with an adverb it uses its monadic k definition.

```
q)6(+)4
'rank
```

We can see this by examining the parse tree:

```
q)parse"6(+)4"
6
(+:;4)
```

What is "+:"?

We know that when parse is used the operators are shown in their k format. Since they are shown in the .q context we can use dictionary reverse lookup to find the meaning:

```
q).q?(+:)
`flip
```

So we can see that in k, when both + and +: are provided a single argument they are equivalent to flip in q:

```
q)d:`c1`c2`c3!(1 2;3 4;5 6)
q)d
c1| 1 2
c2| 3 4
c3| 5 6

q)k)+d
c1 c2 c3
--------
1  3  5
2  4  6

q)k)+:d
c1 c2 c3
--------
1  3  5
2  4  6
```

Note: While in this example `+` and `+:` represent `flip`, this is not true when combined with the each adverb (`'`). In that case, `+:'` would represent `flip each` while `+'` would represent plus-each. Further discussion of this distinction is beyond the scope of this whitepaper.

The monadic functionality of a special character operator can be used in q only if it is wrapped in parentheses:

```

q)+d
'+
q)
q)flip d
c1 c2 c3
--------
1  3  5
2  4  6
q)
q)(+)d
c1 c2 c3
--------
1  3  5
2  4  6
```

Note that when using the reverse lookup on the `.q` context we are slightly hampered by the fact that it is not an injective mapping. The find (`?`) operator will only return the first q keyword matching the k expression. In some cases there is more than one. Instead use the following function:

```
q)qfind:{key[.q]where x~/:string value .q}
q)
q)qfind"k){x*y div x:$[16h=abs[@x];\"j\"$x;x]}"
,`xbar
q)qfind"~:"
`not`hdel
```

We see that `not` and `hdel` are equivalent, although writing the following could be confusing:

```
q)hdel 01001b
10110b
```

Hence in q two different names are provided for clarity.

3.4 Adverbs as Higher Order Functions

An adverb is something which acts on a verb or function to produce a different but related function. This is again easy to see by inspecting the parse tree:

```
q)+/[1 2 3 4]
10
q)parse "+/[1 2 3 4]"
(/;+)
1 2 3 4
```

The first element of the parse tree is `(/;+)` which is itself a parse tree. We know that the first element of any parse tree acts on the remaining arguments, hence `/` (the adverb over) is acting on `+` to produce a new function which sums the elements of a list.

For more detail on using adverbs please refer to Edition 11 of the Technical Whitepaper series "Efficient Use of Adverbs"

4 FUNCTIONAL QUERIES

Alongside each qSql query we also have the equivalent functional forms. These are especially useful for programmatically generated queries, such as when column names are dynamically queried.

```
?[t;c;b;a] // select and exec
![t;c;b;a] // update and delete
```

(where `t` is a table, `a` is a dictionary of aggregates, `b` is a dictionary of group-bys and `c` is a list of constraints in the portable parse tree format).

http://code.kx.com/wiki/JB:QforMortals/queries_q_sql#Functional_Forms_of_select_and_update

The q interpreter parses the syntactic forms of `select`, `exec`, `update` and `delete` into their equivalent functional forms. Therefore there is no performance difference between a free form query and a functional one.

4.1 Issues Converting to Functional Form

To convert a select statement to a functional form one may attempt to apply the `parse` function to the query string:

```
q)parse "select sym,price,size from trade where price>50"
?
`trade
,,(>;`price;50)
0b
`sym`price`size!`sym`price`size
```

As we know, `parse` produces a parse tree and since some of the elements may themselves be parse trees we can't immediately take the output of `parse` and plug it into the form `?[t;c;b;a]`. After a little playing around with the result of parse you might eventually figure out that the correct functional form is as follows:

```
q)funcQry:?[`trade;enlist(>;`price;50);0b;`sym`price`size!
`sym`price`size]
q)
q)strQry:select sym,price,size from trade where price>50
q)
q)funcQry~strQry
1b
```

This, however, becomes more difficult as the query statements become more complex:

```
q)parse "select count i from trade where 140>(count;i) fby sym"
?
`trade
,,(>;140;(k){@[(#y)#x[0]0#x
1;g;:;x[0]'x[1]g:.=y]};(enlist;#:;`i);`sym))
0b
(,`x)!,(#:;`i)
```

In this case, it is not obvious what the functional form of the above query should be, even after applying `parse`.

There are 3 issues with this parse-and-"by eye" method to convert to the equivalent functional form. We will cover these in the next 3 subsections.

4.1.1 Parse Trees and `eval`

The first issue with passing a select statement to `parse` is that each returned element is in unevaluated form. As discussed in section 3.2, simply applying `value` to a parse tree does not work. However, if we evaluate each one of the arguments fully (then there are no nested parse trees) we could then apply `value` to the result:

```
q)eval each parse "select count i from trade where 140>(count;i) fby
sym"
?
+`sym`time`price`size!(`VOD`IBM`BP`VOD`IBM`IBM`HSBC`VOD`MS..
,(>;140;(k){@[(#y)#x[0]0#x
1;g;:;x[0]'x[1]g:.=y]};(enlist;#:;`i);`sym))
0b
(,`x)!,(#:;`i)
```

The below equivalence holds for a general qSql query provided as a string:

```
q)value[str]~value eval each parse str
1b
```

In particular:

```
q)str:"select count i from trade where 140>(count;i) fby sym"
q)
q)value[str]~value eval each parse str
1b
```

In fact, since within the functional form we can refer to the table by name we can make this even clearer. Also, the first element in the result of `parse` applied to a select statement will always be `?` (or `!` for a delete/update query) which cannot be evaluated any further, therefore we don't need to apply `eval` to it.

```
q)pTree:parse str:"select count i from trade where 140>(count;i) fby
sym"
q)@[pTree;2 3 4;eval]
?
`trade
,(>;140;(k){@[(#y)#x[0]0#x
1;g;:;x[0]'x[1]g:.=y]};(enlist;#:;`i);`sym))
0b
(,`x)!,(#:;`i)
q)value[str] ~ value @[pTree;2 3 4;eval]
1b
```

4.1.2 Variable representation in Parse Trees

Recall in a parse tree a variable is represented by a symbol containing its name. So to represent a symbol or a list of symbols, you must use `enlist` on that expression. `enlist` is represented by the comma operator in k:

```
q)parse"3#`a`b`c`d`e`f"
#
3
,`a`b`c`d`e`f
q)(#;3;enlist `a`b`c`d`e`f)~parse"3#`a`b`c`d`e`f"
1b
```

This causes a problem since as discussed in section 3.3 the monadic functionality of special character operators is disabled in q unless wrapped in parentheses:

```
q),`A
',                //error
q)(,)`A           //operators in brackets parsed by k interpreter
,`A
q)type (,)`A
11h
```

Which means the following isn't a valid q expression and so returns an error:

```
q)(#;3;,`a`b`c`d`e`f)
',
```

In the parse tree we receive we need to somehow distinguish between monadic “,” (which we want to replace with `enlist)` and the dyadic join operator (which we want to leave as is).

4.1.3 Explicit definitions in .q are shown in full

The `fby` in the select statement above is represented by its full k definition:

```
q)parse "fby"
k){@[(#y)#x[0]0#x 1;g;:;x[0]'x[1]g:.=y]}
```

While using the k form generally isn’t a problem from a functionality perspective, it does however make the resulting functional statement difficult to read.

5 THE SOLUTION

We will build a function to automate the process of converting a select statement into its respective functional form.

This function, `buildSelect`, will return the functional form as a string:

```
q)buildSelect "select count i from trade where 140>(count;i) fby
sym"
"?[trade;enlist(>;140;(fby;(enlist;count;`i);`sym));0b;(enlist`x)!
enlist (count;`i)]"
```

When executed it will always return the same result as the select statement from which it is derived:

```
q)str:"select count i from trade where 140>(count;i) fby sym"
q)
q)value[str]~value buildSelect str
1b
```

And since the same logic applies to `exec`, `update` and `delete` it will be able to convert to their corresponding functional forms also.

To build this function we will solve the 3 issues outlined in section 4.1.

The first issue, described in section 4.1.1, where some elements returned by the `parse` command may themselves be parse trees is easily resolved by applying `eval` to the individual elements

Section 4.1.2 highlighted an issue with the monadic use of “,”. We want to replace this with the q function `enlist`. To do this we define a function that traverses the parse tree and detects if any element is an enlisted list of symbols or an enlisted single symbol. If it does we will replace this with a string representation of `enlist` instead of “`,`”.

```
q)ereptest:{((0=type x) & (1=count x) & (11=type first x)) |
(11=type x) & (1=count x)};      //returns a boolean

q)ereplace:{"enlist",.Q.s1 first x};

q)funcEn:{$[ereptest x;ereplace x;0=type x;.z.s each x;x]};
```

Before we replace the element we first need to check it has the correct form. We need to test if it is:

1) An enlisted list of syms. A noun will have type 0h, count 1 and type of the first element will be 11h if and only if that noun is an enlisted list of syms.

   or

2) An enlisted single sym. A noun will have type 11h and count 1 if and only if that noun is an enlisted single symbol.

The `ereptest` function above performs this check, with `ereplace` performing the replacement. **Note** that `.Q.s1` is dependent on the size of the console so make it larger if required.

Since we are going to be checking a parse tree which may be made of nested parse trees of an arbitrary depth we need a way of checking all the elements down to the base level.

We observe that a parse tree is a general list, and therefore of type 0h. This knowledge combined with the use of `.z.s` allows us to recursively scan a parse tree. The logic goes: if what you have passed into `funcEn` is a parse tree then reapply the function to each element.

To illustrate we examine the following select statement:

```
q)show pTree:parse "select from trade where sym like \"F*\",not
sym=`FD"
?
`trade
,((like;`sym;"F*");(~:;(=;`sym;,`FD)))
0b
()
q)
q)x:eval pTree 2          //apply eval to where clause
```

Consider the where clause in isolation

```
q)x                       //a 2 element list of where clauses
(like;`sym;"F*")
(~:;(=;`sym;,`FD))
q)
q)funcEn x
(like;`sym;"F*")
(~:;(=;`sym;"enlist`FD"))
```

Similarly we create a function which will replace k functions with their q equivalents in string form, thus addressing the issue raised in section 4.1.3.

```
q)kreplace:{[x] $[`=qval:.q?x;x;string qval]};
q)
q)funcK:{$[0=t:type x;.z.s each x;t<100h;x;kreplace x]};
```

Running these functions against our where clause, we see the k representations being converted to q:

```
q)x
(like;`sym;"F*")
(~:;(=;`sym;,`FD))
q)
q)funcK x
(like;`sym;"F*")
("not";(=;`sym;,`FD))        //replaced ~: with "not"
```

Next, we make a slight change to `kreplace` and `ereplace` and combine them:

```
kreplace:{[x] $[`=qval:.q?x;x;"~~",string[qval],"~~"]};
ereplace:{"~~enlist",(.Q.s1 first x),"~~"};

q)funcEn funcK x
(like;`sym;"F*")
("~~not~~";(=;`sym;"~~enlist`FD~~"))
```

The double tilde here is going to act as a tag to allow us to differentiate from actual string elements in the parse tree. This allows us to drop the embedded quotation marks at a later stage inside the `buildSelect` function:

```
q)ssr/[;("\"~~";"~~\"");("";"")] .Q.s1 funcEn funcK x
"((like;`sym;\"F*\");(not;(=;`sym;enlist`FD)))"
```

thus giving us the correct format for the where clause in a functional select. By applying the same logic to the rest of the parse tree we can create the `buildSelect` function.

```
q)buildSelect "select from trade where sym like \"F*\",not sym=`FD"
"?[trade;((like;`sym;\"F*\");(not;(=;`sym;enlist`FD)));0b;()]"
```

One thing to take note of is that since we use reverse lookup on the `.q` namespace and only want one result we occasionally get the wrong keyword back.

```
q)buildSelect "update tstamp:ltime tstamp from z"
"![z;();0b;(enlist`tstamp)!enlist (reciprocal;`tstamp)]"
q)
q).q`ltime
%:
q).q`reciprocal
%:
```

These instances are rare and the developer should be able to spot when this occurs. Of course the functional form will still work as expected but could confuse future readers of the code.

5.1 5th and 6th Parameters

Functional select also comes with optional 5th and 6th parameters:
http://code.kx.com/wiki/Reference/select

We also cover these with the `buildSelect` function:

```
q)buildSelect "select[10 20] from trade"
"?[trade;();0b;();10 20]"                    //5th parameter included
```

The 6th parameter is a column and direction to order the results by; use < for ascending and > for descending.

```
q)parse"select[10;<price] from trade"
?
`trade
()
0b
()
10
,(<:;`price)

q).q?(<:;>:)
`hopen`hclose

q)qfind each ("<:";">:")      //qfind defined in section 3.3
hopen
hclose
```

We see that the k function for the 6th parameter of the functional form is `<:` or `>:` depending on whether we want the table sorted ascending or descending. At first glance this appears to be `hopen`/`hclose`. In fact in earlier versions of q, `iasc` and `hopen` were equivalent (as were `idesc` and `hclose`). However the definitions of `iasc` and `idesc` were altered to throw a `` `rank`` error if not provided a list as an argument.

```
q)iasc
k){$[0h>@x;'`rank;<x]}

q)idesc
k){$[0h>@x;'`rank;>x]}
q)
q)iasc 7
'rank
```

Since the columns of a table will always be lists, whether the functional form uses the old or new version of `iasc`/`idesc` is irrelevant.

The buildSelect function handles the 6th parameter as a special case so will produce iasc/idesc as appropriate.

```
q)buildSelect "select[10 20;>price] from trade"
"?[trade;();0b;();10 20;(idesc;`price)]"
```

The full buildSelect function code is provided in the appendix.

6 CONCLUSION

This paper has investigated how statements in q are evaluated by the parser. To do this we examined the relationship between q and k, and explained why parse trees generated by the parse function contain k code. We have then used this understanding to explain how qSql statements can be converted into their equivalent and more powerful functional form.

In order to further help those learning how to construct functional statements, and also as a useful development tool, we have created a function which will correctly convert a qSql query into the q functional form. This would be useful for checking that a functional statement is formed correctly or to help with a particularly tricky query.

All tests were run using kdb+ version 3.2 (2015.01.14).

7 APPENDIX

```
\c 30 200
tidy:{ssr/[;("\"~~";"~~\"");("";"")] $[","=first x;1_x;x]};
strBrk:{y,(";" sv x),z};

//replace k representation with equivalent q keyword
kreplace:{[x] $[`=qval:.q?x;x;"~~",string[qval],"~~"]};
funcK:{$[0=t:type x;.z.s each x;t<100h;x;kreplace x]};

//replace eg ,`FD`ABC`DEF with "enlist`FD`ABC`DEF"
ereplace:{"~~enlist",(.Q.s1 first x),"~~"};
ereptest:{((0=type x) & (1=count x) & (11=type first x)) | ((11=type x)
&(1=count x))};
funcEn:{$[ereptest x;ereplace x;0=type x;.z.s each x;x]};

basic:{tidy .Q.s1 funcK funcEn x};

addbraks:{"(",x,")"};

//where clause needs to be a list of where clauses, so if only one where
clause need to enlist.
stringify:{$[(0=type x) & 1=count x;"enlist ";""],basic x};

//if a dictionary apply to both, keys and values
ab:{$[(0=count x) | -1=type x;.Q.s1 x;99=type x;(addbraks stringify key x
),"!",stringify value x;stringify x]};

inner:{[x]
    idxs:2 3 4 5 6 inter ainds:til count x;
    x:@[x;idxs;'[ab;eval]];
    if[6 in idxs;x[6]:ssr/[;("hopen";"hclose");("iasc";"idesc")] x[6]];
    //for select statements within select statements
    x[1]:$[-11=type x 1;x 1;[idxs,:1;.z.s x 1]];
    x:@[x;ainds except idxs;string];
    x[0],strBrk[1_x;"[";"]"]
    };

buildSelect:{[x]
      inner parse x
      };
```