it's about time

Technical Whitepaper

Efficient Use of Adverbs

Author:

Conor Slattery is a Financial Engineer who has designed kdb+ applications for a range of asset classes. Conor is currently working with a New York based investment firm, developing kdb+ trading platforms for the US equity markets.

TABLE OF CONTENTS

INTRODUCTION

In addition to the large number of built in functions and the ability to create your own functions quickly and easily, kdb+ provides adverbs which can alter function and verb behavior to improve efficiency and keep code concise. Employing adverbs correctly can bypass the need for multiple loops and conditionals with significant performance enhancements.

This whitepaper provides an introduction to the basic use of the different adverbs available in kdb+ along with examples of how they differ when applied to monadic, dyadic and higher valence functions. It also covers how adverbs can be combined to further extend the functionality of the built in functions. Common use cases, such as using adverbs for recursion and using adverbs to modify nested columns in a table, are looked at in more detail. These examples provide solutions to common problems that are encountered when building systems in kdb+ and demonstrate the range of situations where adverbs can be used to achieve the desired result.

All tests were run using kdb+ version 3.1 (2013.08.09)

NOTE: Functions in kdb+ can be one of four basic types. These are lambdas (type 100h), primitives (types 101h-103h), projections (type 104h) and compositions (type 105h). In general, the effects of an adverb on a function will depend only on the number and type of the parameters passed to it, not on the type of the function itself. For this reason, we will not distinguish between the four types of functions listed above when talking about adverbs.

1 BASIC USE OF ADVERBS WITH FUNCTIONS

There are six different adverbs, each of which will either modify the functionality of a function, modify the way a function is applied over its parameters or in some cases make no changes at all. Understanding the basic behavior of each adverb and how this behavior varies based on the valence of the underlying function is key for both writing and debugging q code. Note that some adverbs may not be defined for certain types of functions e.g. each-right and each-left will return an error if applied to monadic functions.

Symbol	Name
'	Each-both
':	Each-prior
/	Over
\	Scan
/:	Each-right
\:	Each-left

Table 1: List of adverbs in kdb+

1.1 Monadic functions

1.1.1 Each-Both

The each-both adverb, when used with a monadic function will apply the function to each element of a list.

```
q) type'[(1;2h;3.2)]
-7 -5 -9h
```

If the parameter is atomic, then the each-both adverb will have no effect on the function.

The same behavior described in the above example can also be achieved using the `each` function. This is a dyadic function which will take the left argument, in this case the `type` function, and apply it to each element in the right argument, the list.

```
q) parse "each"
k){x'y}

q) type each (1;2h;3.2)
-7 -5 -9h
```

1.1.2 Each-Prior

The each-prior adverb with a monadic function is used to apply the function to each element of a list, using slave threads when available. Slave threads can be set using the `-s` command line parameter. In the event that no slave threads are set each-prior behaves identically to the `each` function. This adverb is used in the definition of the `peach` function.

```
q) parse "peach"
k){x':y}

q) type peach (3;4.3;2h)
-7 -9 -5h
```

1.1.3 Over

The over adverb, when used with a monadic function will apply the function recursively i.e. the result of the function is calculated repeatedly with the result of one iteration being used as the parameter of the next iteration. This is covered in more detail in Section 3.

```
q) {2*x}/[10;2]
2048
```

1.1.4 Scan

The scan adverb is the same as the over adverb, but it will return the result of every iteration, instead of just the result of the final iteration. This is covered in more detail in Section 3.

```
q) {2*x}\[10;2]
2 4 8 16 32 64 128 256 512 1024 2048
```

1.2 Dyadic functions

1.2.1 Each-Both

The each-both adverb, when used with a dyadic function, will apply the function to each element of the two arguments passed to the function. One, or both of the parameters passed to the function may be atoms. If both arguments are atoms then the adverb will have no effect.

```
q) 1 ~' 1
1b
```

In the case that one of the arguments is an atom and the other is a list then the atom will act as if it is a list of the same length as the non-atom argument.

```
q) 1 ~' 1 2 3
100b
```

If both parameters are lists then they must be of the same length.

```
q) 1 2 3 in' (1 2 3;3 4 5)
'length

q) 1 2 3 in' (1 2 3;3 4 5;5 6 7)
100b
```

1.2.2 Each-Prior

The each-prior adverb, when used with a dyadic function, will apply the function to each element of a list and the previous element. It is equivalent to taking each adjacent pair in the list and applying the dyadic function to each of them. The result will be a list of the same length as the original list passed to the function. A common use of this is in the `deltas` function.

```
q) parse "deltas"
-':

q)deltas 4 8 3 2 2
4 4 -5 -1 0
```

It can also be useful in tracking down errors within lists which should be identical e.g. the .d files for a table in a partitioned database. The below example uses the `differ` function to check for inconsistencies in .d files. `differ` uses the each-prior adverb and is equivalent to `not ~':`

```
q) parse "differ"
~~':

q){1_date where differ get hsym `$"/mydb/",string[x],"/trade/.d"}
each date
2013.05.03 2013.05.04
```

In this case the values of the .d files are extracted from each partition. The `differ` function, which uses the each-prior adverb, is then used to compare each element in the list pair wise. If a .d file is different to the previous .d file in the list, then that date will be returned by the above statement. The first date returned is dropped as the first element of the list will be compared to -1th element of

the list, which is always null, and so they will never match. For the above example the .d files for the 2013.05.03 and 2013.05.04 partitions are different, and should be investigated further.

The `prior` function uses this adverb and can be used to achieve the same functionality. The example below will return all adjacent pairs of a given list. Note that the first element of the first list in the result will be null.

```
q) parse "prior"
k){x':y}

q) {y,x}prior til 5
  0
0 1
1 2
2 3
3 4
```

1.2.3 Each-Right

The each-right adverb will take a dyadic function and apply the left argument to each element of the right argument. If the left argument is an atom then the each-both adverb will produce the same result as the each-right adverb. However, if the left argument is a list then the each-right adverb must be used.

```
q) "ab",'("cde";"fgh";"ijk")
'Length

q) "ab",/:("cde";"fgh";"ijk")
"abcde"
"abfgh"
"abijk"
```

A useful example which involves this adverb is finding the file handle of each column of a table.

```
q) `:/mydb/2013.05.01/trade,/:key[`:/mydb/2013.05.01/trade]except `.d
`:/mydb/2013.05.01/trade`sym
`:/mydb/2013.05.01/trade`time
`:/mydb/2013.05.01/trade`price
`:/mydb/2013.05.01/trade`size
`:/mydb/2013.05.01/trade`ex
```

The above statement joins the file handle of the table to each element in the list of columns, creating five lists of length two. The each-right adverb can then be used with the inbuilt kdb+ `sv` function to create the file handles of each column.

```
q)` sv/: `:/mydb/2013.05.01/trade,/:key[`:/mydb/2013.05.01/trade]except
`.d
 `:/mydb/2013.05.01/trade/sym
 `:/mydb/2013.05.01/trade/time
 `:/mydb/2013.05.01/trade/price
 `:/mydb/2013.05.01/trade/size
 `:/mydb/2013.05.01/trade/ex
```

1.2.4 Each-Left

The each-left adverb behaves the same way as the each-right adverb, except it applies the right argument to each element of the left argument.

```
q) ("cde";"fgh";"ijk"),\:"ab"
"cdeab"
"fghab"
"ijkab"
```

1.2.5 Over

A dyadic function with an over adverb applied to it can be passed one or two arguments. With one list argument, the first and second elements of the list are passed to the function. The function is then called with the result of the previous iteration as the first parameter and the third element of the original argument list as the second parameter. The process continues in this way for the remaining elements of the argument list.

```
q) */[7 6 5 4 3 2 1]
5040
```

With two arguments, the second one being a list, the function is called with the left argument as its first parameter and the first element of the right argument as the second parameter. Next, the function is called with the result of the previous iteration as the first parameter and the third element as the second parameter. The process continues in this way for the remaining elements of the list.

```
q) {ssr[x] . y}/["hello word." ;(("h";"H") ;(".";"!");("rd" ;"rld"))]
"Hello world!"
```

The `over` function uses this adverb and can be used to achieve the same functionality.

```
q) parse "over"
k){x/y}[k){$["\\"=*x;(system;1_x);-5!x]}]

{x+y} over (1 2 3 4)
10
```

1.2.6 Scan

The scan adverb behaves the same way as the over adverb, but it will return the results of each iteration.

```
q) *\[7 6 5 4 3 2 1]
7 42 210 840 2520 5040 5040

q) {ssr[x] . y}\["hello word." ;(("h";"H") ;(".";"!");("rd" ;"rld"))]
"Hello word."
"Hello word!"
"Hello world!"
```

The `scan` function uses this adverb and can be used to achieve the same functionality.

```
q) parse "scan"
k){x\y}[k){$["\\"=*x;(system;1_x);-5!x]}]

q) {x+y} scan (1 2 3 4)
1 3 6 10
```

1.3 Higher valence functions

1.3.1 Each-Both

The each-both adverb for higher valence functions has the same behavior as the each-both adverb for dyadic functions. Again, all parameters have to be either atoms or lists of uniform length.

```
q) 1 2 3 in' (1 2 3;3 4 5;5 6 7)
100b
```

1.3.2 Over

The over adverb can also be used on higher valence functions.

```
q) ({x+y+z}/)[1;2 3 4;5 6 7]
28
```

In the above example the first iteration will use 1 as the x parameter, 2 as the y parameter and 5 as the z parameter. The result of this iteration will then be passed as the x parameter, with 3 as the y parameter and 6 as the z parameter. The process continues in this manner through the remaining elements of the lists.

For this to work, the 2nd and 3rd arguments in the example above must be either atoms or lists of equal length. If one of the parameters is an atom then that value is used in each iteration. For example, the following two statements are equivalent

```
q) ({x+y+z}/)[1;2 3 4;5]
q) ({x+y+z}/)[1;2 3 4;5 5 5]
```

Since the release of kdb+3.1 (2013.07.07), the exponential moving average of a list can be calculated using the scan adverb. While it was possible to define an exponential moving average function prior to this release, using the new syntax will result in faster execution times.

```
//Function defined using the old syntax
q) ema_old: {{z+x*y}\[first y;1-x;x*y]}
//Function defined using the new syntax
//Will only work in kdb+3.1 2013.07.07 and later releases
q) ema_new:{first[y](1-x)\x*y}
q) t:til 10
q) ema_new[0.1;t]
0
0.1
0.29
0.561
0.9049
1.31441
1.782959
2.304672
2.874205
3.486784

//Functions produce the same results but ema_new is significantly
faster
q) ema_old[0.1;t]~ema_new[0.1;t]
1b
q) t2:til 1000000
q) \t ema_old[0.1;t2]
421
q) \t ema_new[0.1;t2]
31
```

2 COMBINING ADVERBS

Multiple adverbs can be used within the same statement, or even applied to the same function in order to achieve a result which cannot be obtained using only one adverb. In this section, we will take a look at some of the most commonly used and useful examples. While the results produced by these examples might seem confusing initially, by taking each adverb in order and applying it to its monadic, dyadic or higher valence function we can see that the rules outlined in Section 1 are still being followed.

The example below uses both the each-prior adverb and the scan adverb to return the first n rows of Pascal's Triangle.

```
q) pascal:{[numRows] fn:{(+ ':)x,0} ; fn\[numRows;1] };
q) pascal[7]
1
1 1
1 2 1
1 3 3 1
1 4 6 4 1
1 5 10 10 5 1
1 6 15 20 15 6 1
1 7 21 35 35 21 7 1
```

In order to understand what is happening here first look at the definition of the function `fn`. Here the each-prior adverb is applied to the plus operator. This means that this function will return the sum of all adjacent pairs in the argument passed to it. Zero is appended to the end of the argument in order to maintain the final one in each row of the final result.

The scan adverb is applied to the monadic function `fn` in order to use the results of one iteration of this function as the argument of the next iteration. After `numRows` iterations the result of each iteration, along with the initial argument passed to `fn`, will be returned.

A commonly used example of applying multiple adverbs to a function is illustrated in the following piece of code `,/:\:`, which will return all possible combinations of two lists by making use of the each-left and each-right adverbs and applying them to the join "`,`" function. The order of the adverbs will affect the result.

For example:

```
q) raze (1 2 3),/:\:4 5 6
1 4
1 5
1 6
2 4
2 5
2 6
3 4
3 5
3 6

q) raze (1 2 3),\:/:4 5 6
1 4
2 4
3 4
1 5
2 5
3 5
1 6
2 6
3 6
```

To get an idea of how the q interpreter is handling the above statement, note that the following are equivalent:

```
q) (raze (1 2 3),/:\:4 5 6)~ (1,/:4 5 6),(2,/:4 5 6),(3,/:4 5 6)
1b

q) (raze (1 2 3),\:/:4 5 6)~(1 2 3,\:4),(1 2 3,\:5),(1 2 3,\: 6)
1b
```

Another example of joining adverbs is `,//`. This will flatten a nested list repeatedly until it cannot be flattened any more. The two over adverbs in this example are doing different things. The inner over is applied to the dyadic join function, so it joins the first element of the list passed to it to the second, then joins the result to the third element and continues through the remaining elements of the list. (Note that `,/` acts in the same way as the `raze` function)

```
q) ,/[(1 2 3;(4 5;6);(7;8;(9;10;11)))]
1
2
3
4 5
6
7
8
9 10 11

q) raze (1 2 3;(4 5;6);(7;8;(9;10;11)))
1
2
3
4 5
6
7
8
9 10 11
```

This means the `,/` is a monadic function, as it takes a single list as its argument (Like all monadic functions with over or scan applied, it is possible to pass two arguments but it is still treated like a monadic function). When the outer over is applied to this function, it will implement the `,/` function recursively until the result is the same as the input, i.e. the list cannot be flattened any more.

```
q) ,//[(1 2 3;(4 5;6);(7;8;(9;10;11)))]
1 2 3 4 5 6 7 8 9 10 11
```

The each-both adverb can also be combined with itself in order to apply your function to the required level of depth in nested lists.

```
q) lst:(3 2 8;(3.2;6h);("AS";4))

q) type lst
0h

q) type'[lst]
7 0 0h

q) type''[lst]
-7 -7 -7h
-9 -5h
10 -7h

q) type'''[lst]
-7 -7 -7h
-9 -5h
(-10 -10h;-7h)
```

3 USING ADVERBS FOR RECURSION

As explained above, when the over or scan adverbs are used with monadic functions they will operate recursively.

A monadic function with an over or scan adverb may be passed one or two parameters.
If two parameters are used then the first parameter is either:

1. An integer representing the number of iterations to run before stopping
2. A function to which the result of each iteration is passed. If the condition defined by the function in this parameter does not hold, then the scan/over terminates and the result is returned.

```
 //Define a function to calculate a Fibonacci sequence using the over
adverb
 q) fib: {x,sum -2#x}/

//Call the function with an integer as the first parameter
 q) fib[10;1 1]
 1 1 2 3 5 8 13 21 34 55 89 144

 //Call the function with a function as the first parameter
 q) fib[{last[x]<200};1 1]
 1 1 2 3 5 8 13 21 34 55 89 144 233
```

If no exit condition is supplied, the function will continue until one of the two following conditions is detected.

1. The result of the i^{th} iteration is equal to the result of the $(i-1)^{th}$ iteration
2. The result of the i^{th} iteration is equal to the original parameter passed to the function, within tolerance levels if floating point numbers are used.

However, certain loops will not be caught and will result in infinite loops. Consider the function defined below and is illustrated in Fig.1.

```
q) ({3.2*x*(1-x)}\)[30;0.4]
```

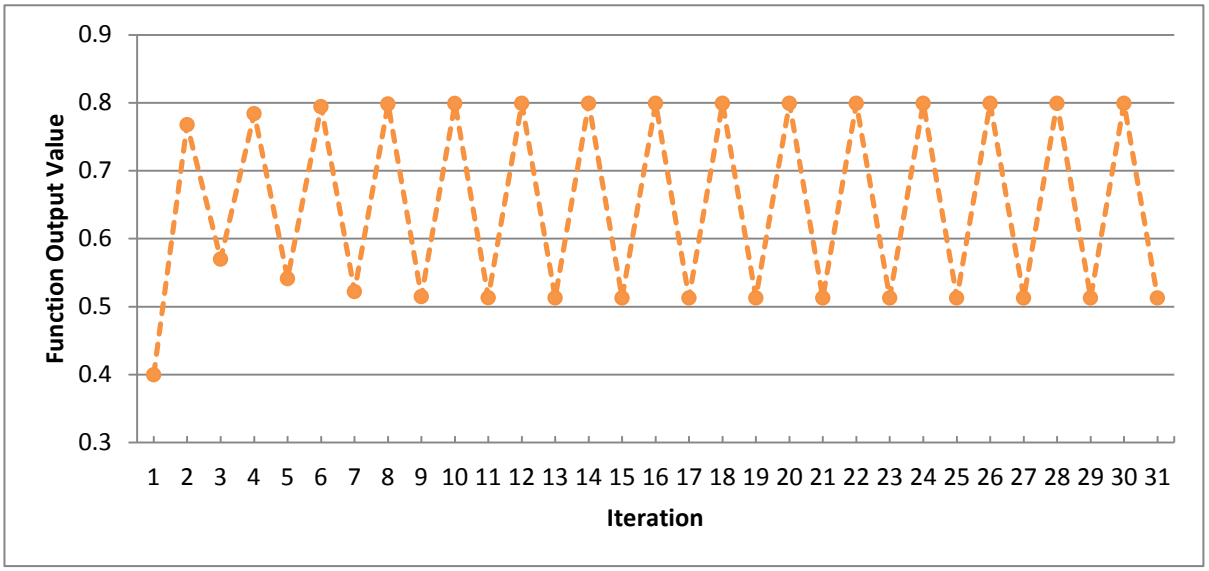

Figure 1: Infinite looping function example

From Fig.1 above, it is evident that this results in a loop with period 2 (at least within floating point tolerance). However, if no exit condition is supplied then the above function will not terminate.

Note: When working with recursive functions, it may be a good idea to set the timeout in your q session via the `\T` command. This will cause any functions to terminate after a set number of seconds and means that infinite loops will not lock your instance indefinitely.

The self function `.z.s` can also be used in recursion, and can allow greater flexibility than using the over or scan adverbs. For example

```
q) {$[0h=type x;.z.s'[x];10h=abs type x;upper x;x]}(`a`n;(1
2;"efd");3;("a";("fes";3.4)))
```

The above function will operate on a list of any structure and data types, changing strings and characters to upper case and leaving all other elements unaltered. Note that when using `.z.s` the function will error out with a `'stack` error message after 2000 loops. This can be seen using the example below:

```
q) {.z.s[0N!x+1]}0
```

No such restriction exists when using the scan or over adverbs, and the `.z.s` function should only be used in cases where it is not possible to use the scan or over adverbs.

4 ADVERBS VS. LOOPS

Many of the built in kdb+ operators have been overloaded to work with atoms, lists or a combination of both. For example, the plus operator can take two atoms, an atom and a list or two lists as arguments. In cases where more control over the execution is desired, or where we are working with user defined functions, either loops or adverbs can be used to achieve the desired results. In almost all cases, adverbs will allow us to create shorter code with lower latency, while also avoiding the creation of unnecessary global variables.

Often the implementation is relatively easy, using each, each-left and each-right to cycle through a list and amend the elements as desired. As an example, if we wanted to check if either of the integers 2 or 3 are present in the lists provided. This can be achieved using a `while` loop:

```
q) {i:0;a:();while[i<count x;a,:enlist any 2 3 in x[i];i+:1];a}(1 2
3;3 4 5;4 5 6)
110b

q)\t:100000 {i:0;a:();while[i<count x;a,:enlist any 2 3 in
x[i];i+:1];a}(1 2 3;3 4 5;4 5 6)
515
```

However, by using adverbs we can create neater, more efficient code:

```
q) any each  2 3 in/: (1 2 3;3 4 5;4 5 6)
110b

q)\t:10000 any each  2 3 in/: (1 2 3;3 4 5;4 5 6)
374
```

Similarly we can use the over adverb to easily deal with situations which would be handled by loops in other, c-like languages. For example suppose you wanted to join a variable number of tables.

```
//Create a list of tables, of random length
q) tlist:{1!flip (`sym;`$"pr",x;`$"vol",x)!(`a`b`c;3?50.0;3?100)}each
string til 2+rand 10

//Join the tables using a while loop
q) {a:([]sym:`a`b`c);i:0;while[i<count[x];0N!a:a lj x[i];i+:1];a}tlist
sym pr0      vol0 pr1      vol1 pr2      vol2
-----------------------------------------------
a   35.2666  53   38.08624 95   1.445859 57
b   19.28851 39   6.41355  50   12.97504 24
c   23.24556 84   13.62839 19   6.89369  46

q)\t:100 {a:([]sym:`a`b`c);i:0;while[i<count[x];0N!a:a lj
x[i];i+:1];a}tlist
101

//Join the tables using the over adverb
q) 0!(lj/)tlist
sym pr0      vol0 pr1      vol1 pr2      vol2
-----------------------------------------------
a   35.2666  53   38.08624 95   1.445859 57
b   19.28851 39   6.41355  50   12.97504 24
c   23.24556 84   13.62839 19   6.89369  46

q)\t:100 0!(lj/)tlist
82
```

5 DEALING WITH NESTED COLUMNS

While it is usually best practice to avoid nested columns, there are some situations where operating on nested data is necessary or may result in lower execution time for certain queries. The main reason for this is that the function `ungroup`, which flattens a table containing nested columns, is computationally expensive, especially when you are only dealing with a subset of the entire table. There are also situations where storing the data in a nested structure makes more sense. For example you may want to use strings, which are lists of characters, instead of symbols, which are atoms, in order to avoid a bloated sym file. For this reason we will now take a look at using adverbs to apply functions to a table as a whole, and to apply functions within a `select` statement.

Adverbs can be used to examine and modify tables. In order to do this, an understanding of how tables are structured is necessary. In kdb+, a table is a list of dictionaries, which are themselves lists which may have a non-integer domain.

This means that we can apply functions to individual elements, just like any other nested list or dictionary structure. For example:

```
q) a:([]a:`a`b`c`d;b:1 2 3 4;c:(1 2;2 3;3 4;4 5))

q) type[a]
98h

q) type'[a]
99 99 99 99h

q) type''[a]
a   b  c
--------
-11 -7 7
-11 -7 7
-11 -7 7
-11 -7 7
```

We can see here that the `type[a]` returns 98h, a table as expected. `type'[a]` returns the type of each element of the list `a`, which are dictionaries. `type''[a]` will find the type of each element in the range of each dictionary in `a`. It returns a list of dictionaries which collapses back to a table showing the type of each field in the table `a`.

```
q) distinct type''[a]
```

In this way, the statement can be used to ensure that all rows of the table are the same type. This is useful if your table contains nested columns, as the `meta` function only looks at the first row of

nested columns. If the table is keyed then the function will only be applied to the non-key columns in this case.

```
q) a:([]a:`a`b`c`d;b:1 2 3 4;c:(1 2;2 3;3 4.;4 5))

q) meta a
c| t f a
-| -----
a| s
b| j
c| J

q) distinct type''[a]
a   b  c
--------
-11 -7 7
-11 -7 9
```

Looking only at the results of `meta`, we may conclude that the column c contains only integer lists. However `distinct type''[a]` clearly shows that the column `c` contains lists of different types, and thus is not mappable. This is a common cause of error when writing to a splayed table.

Dealing with nested data in a table via a `select`/`update` statement often requires the use of adverbs. In order to illustrate this, let us define a table with three columns, two of which are nested.

```
q) tab:([]sym:`AA`BB`CC;time:3#enlist 09:30+til
30;price:{30?100.0}each til 3)
```

Suppose we wanted to find the range of each row. This can be easily done by defining a range function as:

```
q) rng:{max[x]-min[x]}
```

We can then make use of this function within a `select` statement with an each adverb to apply the function to each row of the table.

```
q) select sym, rng'[price] from tab
sym price
------------
AA  96.3872
BB  95.79704
CC  98.31252
```

Suppose instead that we wanted to find the range of a subset of the data available in the table. One way to do this would be to `ungroup` the table and then find the range as follows:

```
q) select rng price by sym from ungroup tab where time within 09:40
09:49
sym| price
---| --------
AA | 77.67457
BB | 80.14611
CC | 67.48254
```

However, it is faster to index into the nested list as this avoids the costly `ungroup` function. First find the index of the prices which fall within our time range:

```
q) inx:where (exec first time from tab) within 09:40 09:49
```

Then use this to index into each price list and apply the `rng` function to the resulting prices.

```
q) select sym, rng'[price@\:inx] from tab
sym inx
------------
AA  77.67457
BB  80.14611
CC  67.48254
```

This offers a significant improvement in latency over using `ungroup`.

```
q)\t:10000 select rng price by sym from ungroup tab where time within
09:40 09:49
198

q)\t:10000 inx:where (exec first time from tab) within 09:40
09:49;select sym, rng'[price@\:inx] from tab
65
```

If the nested lists are not uniform the code needs to be changed to the following:

```
q) inx:where each (exec time from tab) within 09:40 09:49
q) select sym, rng'[price@'inx] from tab
sym inx
------------
AA  77.67457
BB  80.14611
CC  67.48254
```

6 CONCLUSION

This whitepaper provided a summary of the adverbs available in kdb+, showing how it modifies the behavior of different types of functions. It showed, through the use of examples, that by looking at the valence of a function and the adverb being applied to it, the effect of the adverb on the function can be determined. Even more complicated examples using multiple adverbs can be broken down in this manner to understand the behavior.

Certain uses of adverbs, such as the creation of recursive functions and applying adverbs to functions within `select` statements were examined in more detail as these are areas which are often poorly understood and are useful in many situations. Some common uses were looked at in order to demonstrate the ability of adverbs to reduce execution times.

The purpose of this whitepaper was to illustrate how adverbs can be used to easily extend the functionality of built in and user defined functions, allowing for code that takes full advantage of the ability of the q language to process large volumes of data quickly. Correctly using adverbs on data minimizes the amount of manipulation necessary to achieve the desired result, while also allowing for more concise code which is easier to maintain.

All tests were run using kdb+ version 3.1 (2013.08.09)