kx | it's about time

COMPENDIUM OF TECHNICAL WHITE PAPERS

Compendium of Technical White Papers from **Kx**

Contents

Machine Learning

Development Insight

Optimization Techniques

Solutions

it's about time

Technical Whitepaper

1 Machine Learning in kdb+: kNN classification and pattern recognition with q

Author:

Emanuele Melis works for Kx as kdb+ consultant. Currently based in the UK, he has been involved in designing, developing and maintaining solutions for Equities data at a world leading financial institution. Keen on machine learning, Emanuele has delivered talks and presentations on pattern recognition implementations using kdb+.

CONTENTS

1.1 Loading the Dataset

Once downloaded, this is how the dataset looks in a text editor:

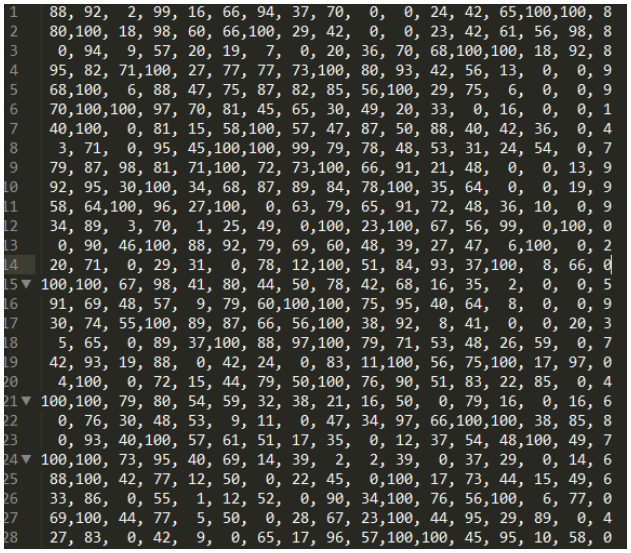


Figure 1. CSV dataset

The last number on the right-hand side of each line is the class label, and the other sixteen numbers are the class attributes; 16 numbers representing the 8 Cartesian coordinates sampled from each handwritten digit.

We start by loading both test and training sets into a q session:

```
q) test:`num xkey flip ((`$'16#.Q.a),`num)!((16#"i"),"c"; ",") 0:`pendigits.tes

q) show test
num| a  b   c  d   e  f  g   h  i   j  k   l  m   n   o   p
---| ------------------------------------------------------
8  | 88 92 2 99 16 66 94 37 70 0  0        24 42 65 100 100
8  | 80 100 18 98 60 66 100 29 42 0  0     23 42 61 56  98
8  | 0  94  9  57  20 19 7   0 20 36 70    68 100 100 18  92
9  | 95 82 71 100 27 77 77 73 100 80 93    42 56  13  0   0
9  | 68 100 6 88 47 75 87 82 85 56 100     29 75  6   0   0
..

q) tra:`num xkey flip ((`$'16#.Q.a),`num)!((16#"i"),"c"; ",") 0:`pendigits.tra

q) show tra
num| a  b   c  d   e   f   g   h   i  j  k   l  m   n  o   p
---| ------------------------------------------------------
8  | 47 100 27 81  57  37  26  0   0 23 56 53 100 90 40  98
2  | 0 89 27 100 42    75  29 45 15 15 37 0 69 2 100 6
1  | 0 57 31 68  72    90  100 100 76 75 50 51 28 25 16  0
4  | 0 100 7 92  5     68  19 45 86 34 100 45 74 23 67  0
1  | 0 67 49 83  100   100 81 80 60 60 40 40 33 20 47  0
..
```

For convenience, the two resulting dictionaries are flipped into tables and keyed on the class attribute so that we can later leverage q-sql and some of its powerful features. Keying at this stage is done for display purposes only. Rows taken from the sets will be flipped back into dictionaries and the class label will be dropped[2] while computing distance metric.

[2] Un-keying and flipping a table into dictionary to drop the first key is more efficient than functionally deleting a column.

The column names do not carry any information and so the first 16 letters of the alphabet are chosen for the 16 integers representing the class attributes; while the class label, stored as character, is assigned a mnemonic tag "*num*".

1.2 Calculating distance metric

As mentioned previously, in a k-NN classifier the distance metric between instances is the distance between their feature arrays. In our dataset, the instances are rows of the *tra* and *test* tables, and their attributes are the columns. To better explain this, we demonstrate with two instances from the training and test set:

```
q) show tra1:1#tra
num| a  b   c  d  e  f  g  h i j  k  l  m   n o  p
---| -----------------------------------------------
8 | 47 100 27 81 57 37 26 0 0 23 56 53 100 90 40 98

q) show tes1:1#test
num| a  b  c d  e  f  g  h  i  j k l  m  n  o   p
---| ----------------------------------------------
8 | 88 92 2 99 16 66 94 37 70 0 0 24 42 65 100 100
```

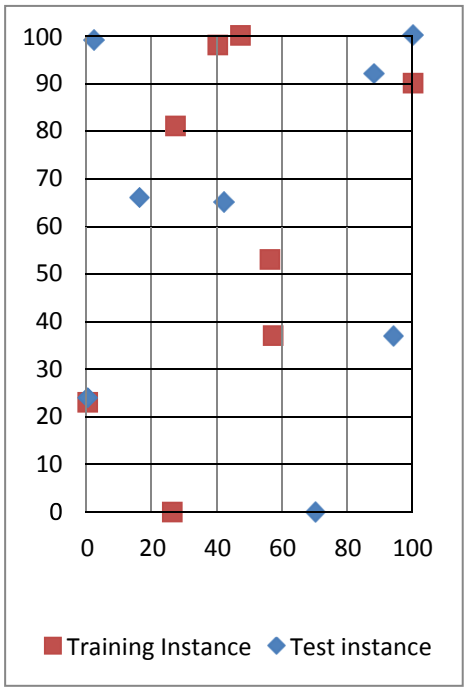


Figure 2-1 tra1 and tes1 point plot

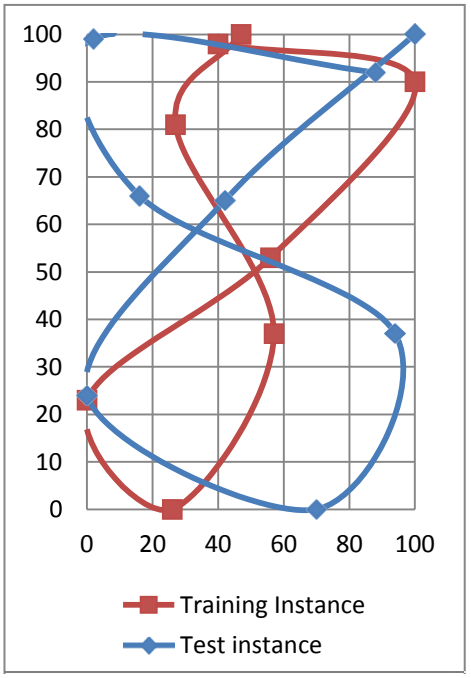


Figure 2-2 tra1 and tes1 visual approximation

Both instances belong to the class digit "8", as per their class labels. However, this is not clear by just looking at the plotted points and the *num* column is not used by the classifier, which will instead calculate the distance between matching columns of the two instances. That is, calculating how far the *a* to *p* columns of *tes1* are from their counterparts in *tra1*. While only an arbitrary measure, the classifier will use these distances to identify the nearest neighbour(s) in the training set and make a prediction.

In q, this will be achieved using a dyadic function whose arguments can be two tables. Using the right adverbs, it is applied column by column, returning one table that stores the result of each iteration in the *a* to *p* columns. The main benefit of this approach is that it relieves the developer from the burden of looping and indexing lists when doing point-point computation.

The metric that will be used to determine the distance between the feature points of two instances is the Manhattan distance $\sum^{k}{}_{i=1} |x_i - y_i|$, and it is calculated as the sum of the absolute difference of their Cartesian coordinates. Using a Cartesian distance metric is intuitive and convenient as the columns in our set represent indeed X or Y coordinates:

```
q) dist:{abs x-y}

q) tra1 dist' tes1
num| a  b c  d  e  f  g  h  i  j  k  l  m  n  o  p
---| ----------------------------------------------
8  | 41 8 25 18 41 29 68 37 70 23 56 29 58 25 60 2
```

Now that the resulting table represents the distance metric between each attribute of the two instances, we can sum all the values and obtain the distance between the two instances in the feature space:

```
q) sums each tra1 dist' tes1
num| a  b  c  d  e   f   g   h   i   j   k   l   m   n   o   p
---| ------------------------------------------------------------
8 | 41 49 74 92 133 162 230 267 337 360 416 445 503 528 588 590
```

The q verb *sums* adds up the columns, from left to right, and the last column, *p*, holds the total: 590, which represents the Manhattan distance between *tra1* and *tes1*. Expanding to run against the whole training set, *tra,* will return the list of Manhattan distances between *tes1* and all the instances in the set, upon which the classifier will try to make its prediction. Calculating those distances can be done trivially using the adverb *each*.

Calculating the distance metric is possibly the heaviest computing step of a k-NN classifier, and *ts* can be used to compare the performance between the use of the *left/right* variations of *each*, capturing their total execution time after an arbitrary high number of iterations (5000 in the example was enough to show a difference). As both sets are keyed on *num*, and *tra* contains more than one instance, a change of paradigm and adverb is necessary:

- Un-key and remove the *num* column from *tes1*
- Update the adverb so that *dist* gets applied to all rows of the *tra* table:

```
q)\ts:5000 tra dist\: 1_flip 0!tes1
203770 5626544

q)\ts:5000 (1_flip 0!tes1) dist/: tra
167611 5626544
```

Keeping *tes1* as the left argument while using the *each-right* adverb makes the execution a little more time efficient due to how *tes1* and *tra* are serialized and how *tra* is indexed. Moreover, *each-right* makes the order of the operations clearer, we are calculating the distance between the left argument (validation instance) and each row on the table in the right argument (training set).

In q, lambda calculus is supported and functions are “first class citizens”:

```
q) {sums each (1_x) dist/: tra} flip 0!tes1
num| a  b   c   d   e   f   g   h   i   j   k   l   m   n   o   p
---| ----------------------------------------------------------
8  | 41 49  74  92  133 162 230 267 337 360 416 445 503 528 588 590
2  | 88 91  116 117 143 152 217 225 280 295 332 356 383 446 446 540
1  | 88 123 152 183 239 263 269 332 338 413 463 490 504 544 628 728
4  | 88 96  101 108 119 121 196 204 220 254 354 375 407 449 482 582
1  | 88 113 160 176 260 294 307 350 360 420 460 476 485 530 583 683
6  | 12 20  106 106 139 147 224 234 304 320 357 381 412 461 541 621
4  | 88 96  97  124 134 165 174 176 206 277 350 423 446 462 496 596
..
```

Note, there is no argument validation done within the lambda (this has minimal memory footprint and compute cost), which means *x* must be a dictionary with the same keys as the table *tra*, *num* column excluded.

The first step of the classifier is complete, let's store the lambda into a variable called *apply_dist*. Columns *a* to *o* are not needed and can be removed. *P* is renamed "*distance"*:

```
q) apply_dist:{[d;t] select num,distance:p from sums each t dist/: d}
```

Projecting *apply_dist* into a monadic function allows for further optimization. As *dist* expects a dictionary, the test instance can be passed as a second parameter: a dictionary razing it after removing the *num* column; or as a table using the adverb *each* after removing the column *num*. The returned result will be a list of tables if the parameter is passed as a table of length>1, otherwise, a single table.

```
q) apply_dist[tra;]each delete num from tes1
+`num`distance!("8214164050985973392251586404818570935492756336325431014532O6..

q) apply_dist[tra]each 2#delete num from tes1
+`num`distance!("8214164050985973392251586404818570935492756336325431014532O6..
+`num`distance!("8214164050985973392251586404818570935492756336325431014532O6..

q) apply_dist[tra;]raze delete num from tes1
num distance
------------
8   590
2   540
1   728
4   582
..
```

1.3 k-Nearest Neighbors and prediction

The classifier is almost complete. As the column *p* now holds the distance metric between the test instance and each training instance, to validate its behaviour we shall find if there are any instances of *tra* where *distance*<590.

As the result of the *apply_dist* is a table, we use q-sql to keep the code readable, without compromising performance. The benefits of this approach will be clearer at the end of this chapter after having analysed the possible implementations for k=1 and k>1.

1.3.1 Nearest Neighbor k=1

If k=1 the prediction is the class of the nearest neighbour - the instance with the smallest distance in the feature space. As all distance metrics are held in the column *distance*, we want to "**select** the class_label of the row **from** the result of applying the distance metric **where** the **distance** is the **minimum** distance in the whole table". In q-sql:

```
q) select Prediction:num, distance from apply_dist[tra;]raze[delete num from tes1] where
distance=min distance
Prediction distance
------------------
8          66

q) select from tra where i in exec i from apply_dist[tra;]raze[delete num from tes1] where
distance=min distance
num| a  b  c d   e  f  g  h  i  j k l  m  n  o   p
---| -----------------------------------------------
8 | 76 85 0 100 16 63 81 31 69 0 6 25 34 64 100 95
```

Querying the virtual column *i*, we can find the row index of the nearest neighbour used for the prediction. Plotting this neighbour we see several points now overlap, while the distance between non-overlapping points is definitely minimal compared to the previous example (*Figure 2-2*):

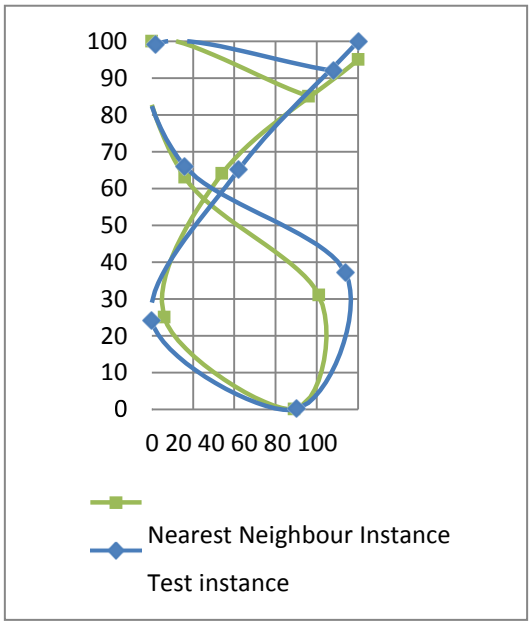


Figure 3 tes1 and its nearest neighbour

Another approach to finding the k-nearest neighbour is to sort the result of *apply_dist* in **ascending order by distance**, this way, the first k rows are the k-nearest neighbors. Taking only the **first row** is equivalent to selecting the nearest neighbour.

```
q) select Prediction:num from 1#`distance xasc apply_dist[tra;]raze delete num from tes1
Prediction
----------
8
```

Or instead limit to the first row by using the row index:

```
q) select Prediction:num from `distance xasc apply_dist[tra;]raze[delete num from tes1] where
i<1
Prediction
----------
8
```

Sorting *distance* has the side effect of applying the *sorted* attribute to the column, this will allow kdb+ to use a different search algorithm on that column, for the benefit of speed, with very little trade off.

1.3.2 k>1

For k>1, the prediction is instead the predominant class among the k-nearest neighbors. Tweaking the above example so that 1 is abstracted into the parameter k:

```
q) k:3; select Neighbour:num,distance from k#`distance xasc apply_dist[tra;]raze delete num from
tes1
Neighbour distance
------------------
8         66
8         70
8         75

q) k:5; select Neighbour:num,distance from k#`distance xasc apply_dist[tra;]raze delete num from
tes1
Neighbour distance
------------------
8         66
8         70
8         75
8         92
8         94
```

At this point, knowing which results are the k nearest neighbors, the classifier can be built further to have class “prediction” functionality. It is common to use odd values of k to prevent uncertainty when determining a class, however it can still happen that there is no clear predominant class amongst the neighbors. In this scenario one can choose to either drop the classification for the uncertain sample, add a weight to their distance metric, or fall back to use smaller values of k.
The steps needed to build the k-NN circle can be summarized into a function *get_nn*:

```
q) get_nn: {[k;d] select nn:num,distance from k#`distance xasc d}
```

1.3.3 4.3 Prediction test

The next step is to make a prediction based on the classes identifies in the k-NN circle. This will be done by counting the number of occurrences of each class among the k-nearest neighbors and picking the one with the highest count.

As the k-NN circle returned by *get_nn* is a table, the q-sql *fby* construct can be used to apply the aggregating function *count* to the virtual column *i*, counting how many rows per each *nn* class label are in the in k-NN circle table, and compare it with the highest count:

```
q) predict: {1#select Prediction:nn from x where ((count;i)fby nn)=max (count;i)fby nn}
```

For k>1, it is possible that *fby* can return more than one instance, should there not be a prevailing class. Given that *fby* returns entries in the same order they are aggregated, class labels are returned in the same order they are found. Thus, designing the classifier to take only the first row of the results has the side effect of defaulting the behaviour of *predict* to k=1. Consider this example:

```
q) foo1: {   select Prediction:nn from x where ((count;i)fby nn)=max (count;i)fby nn};

q) foo2: { 1#select Prediction:nn from x where ((count;i)fby nn)=max (count;i)fby nn};

q) (foo1;foo2)@\: ([] nn:"28833"; distance: 20 21 31 50 60) // Dummy table, random values
+(,`Prediction)!,"8833"     // foo1, no clear class
+(,`Prediction)!,,"2"       // foo2, default to k=1; take nearest neighbour
```

Let's now test *predict* with k=5 and *tes1*:

```
q) predict get_nn[5;] apply_dist[tra;]raze delete num from tes1
Prediction
----------
8
```

Spot on!

1.4 Accuracy checks

At this stage, we can feed the classifier with the whole test set, enriching the result of *predict* with a column *Test* for the class label of the test instance, a column *Hit* which will be true when prediction and class label match:

```
q) test_harness:{[d;k;t]
     select Test:t`num, Prediction, Hit:Prediction=' t`num from
     predict get_nn[k] apply_dist[d] raze delete num from t
   };
```

1.4.1 Running with k=5

```
q) R5:test_harness[tra;5;] peach 0!test

q) R5
+`Test`Prediction`Hit!(,"8";,"8";,1b)
+`Test`Prediction`Hit!(,"8";,"8";,1b)
+`Test`Prediction`Hit!(,"8";,"8";,1b)
+`Test`Prediction`Hit!(,"9";,"9";,1b)
..
```

As the result of *test_harness* is a list of predictions, it needs to be razed into a table to extract the overall accuracy stats. The accuracy measure will be the number of hits divided by the number of predictions made for the correspondent validation class:

```
q) select Accuracy:sum[Hit]%count i by Test from raze R5
Test| Accuracy
----| ---------
0   | 0.9614325
1   | 0.9505495
2   | 0.9945055
3   | 0.9940476
4   | 0.978022
5   | 0.9761194
6   | 1
7   | 0.9532967
8   | 0.9970238
9   | 0.9672619
```

1.4.2 Running with k=3

```
q) R3:test_harness[tra;3;] peach 0!test

q) select Accuracy:sum[Hit]%count i by Test from raze R3
Test| Accuracy
----| ---------
0   | 0.9641873
1   | 0.9587912
2   | 0.9917582
3   | 0.9940476
4   | 0.9807692
5   | 0.9701493
6   | 1
7   | 0.967033
8   | 0.9970238
9   | 0.9583333
```

1.4.3 Running with k=1

```
q) R1:test_harness[tra;1;] peach 0!test

q) select Accuracy:sum[Hit]%count i by Test from raze R1
Test| Accuracy
0   | 0.9614325
1   | 0.9505495
2   | 0.9917582
3   | 0.9940476
4   | 0.978022
5   | 0.961194
6   | 1
7   | 0.956044
8   | 0.9940476
9   | 0.9583333
```

1.5 Further Approaches

1.5.1 Use slave threads

Testing and validation phases will benefit significantly from the use of slave threads in kdb+, applied through the use of the *peach* function:

s 0 (0 slaves) - Run time: ~ 47s

```
q)\ts test_harness[tra;1;] each 0!test
47225 2946880

q) 47225 % 1000
47.225
```

-s 4 (4 slaves) - Run time: ~24s

```
q)\ts test_harness[tra;1;] peach 0!test
24431 744656

q) 24431 % 1000
24.431
```

1.5.2 Code optimization

The function *dist* wraps a series of operations to calculate the distance metric, this is a convenient approach for non-complex metrics and testing as it can be changed on the fly without having to change *apply_dist*'s implementation. However, it is executed 16 times each row in *d*, which makes it worth testing a more performant approach. As mentioned earlier q is an array processing language, doing the arithmetic on the table directly will remove levels of indirection and leverage internal optimization with a net gain in performance:

```
q)\ts:250 {[d;t] select num,distance:p from sums each abs t dist/: d}[tra;] raze delete num from
tes1
3335 2314720 // 13.34 per run

q)\ts:250 {[d;t] select num,distance:p from sums each abs t -/: d}[tra;] raze delete num from
tes1
3002 2314720 // 12.00 per run
```

The speed difference between the two approaches will become obvious using more complex metrics as examined in section 6.3.

1.5.3 Euclidean or Manhattan Distance?

The Euclidean distance metric $\sqrt{\sum_{i=1}^{k}(x_i - y_i)^2}$ can be intuitively implemented as:

```
q) apply_dist:{[d;t] select num,distance:p from sqrt sums each t {(x-y) xexp 2}/: d}
```

However, following the approach in section 6.2, the implementation can be further optimized by squaring each difference instead of using *xexp*:

```
q)\ts:1000 r1:{[t;d] t {(x-y)xexp 2}/: d}[tra;]raze delete num from tes1
550727 19530080 // Slower and requires more memory

q)\ts:1000 r2:{[t;d] t {x*x:x-y}/: d}[tra;]raze delete num from tes1
  5723  8917312
```

The function *xexp* has two caveats:

- for an exponent of 2 it is faster to *not* use xexp and, instead, multiply the base by itself
- it returns a float, meaning each cell in the result set will increase in size by 4 bytes

```
q) min over (=) . (r1;r2)
1b   // Same values

q) (~) . (r1;r2)
0b   // Not the same objects

q) type @''' value each (r1;r2)
+`a`b`c`d`e`f`g`h`i`j`k`l`m`n`o`p!(-9 -9 -9 -9 -9 -9 -9 -9 -9h;-9 -9 -9 -9..
+`a`b`c`d`e`f`g`h`i`j`k`l`m`n`o`p!(-6 -6 -6 -6 -6 -6 -6 -6 -6h;-6 -6 -6 -6..

q) -22!'(r1;r2)
1467 827
```

Another approach would be to square the result of *t -/: d*. Despite being as fast as squaring each cell, its memory utilization depends on the size of the training set. With *tra* it uses almost 3 times as much memory:

```
q)\ts:1000 r3:{[t;d] t*t:t -/: d}[tra;]raze delete num from tes1
 5947 25177360

q)\ts:1000 r2:{[t;d] t {x*x:x-y}/: d}[tra;]raze delete num from tes1
 5723  8917312

q) min over (=) . (r2;r3)
1b

q) (~) . (r2;r3)
1b
```

Choosing the optimal implementation, we can benchmark against the full test set:

```
q) apply_dist:{[d;t] select num,distance:p from sqrt sums each t {x*x:x-y}/: d}

q)\ts R1:test_harness[tra;1;] peach 0!test
31584 744704 // ~7 seconds slower than the Manhattan distance benchmark

q) select Accuracy:sum[Hit]%count i by Test from raze R1
Test| Accuracy
----| ---------
0   | 0.9752066
1   | 0.9587912
2   | 0.9945055
3   | 0.9910714
4   | 0.9752747
5   | 0.9701493
6   | 1
7   | 0.956044
8   | 0.9970238
9   | 0.9583333
```

How does this compare to the Manhattan distance implementation?

```
q) select Accuracy_Manhattan:(avg;med)@\: Accuracy from select Accuracy:sum[Hit]%count i by Test
from raze R1
Accuracy_Manhattan
------------------
0.9745429
0.9697272

q) select Accuracy_Euclidean:(avg;med)@\: Accuracy from select Accuracy:sum[Hit]%count i by Test
from raze R1
Accuracy_Euclidean
------------------
0.97764
0.9752407
```

1.6 Conclusions

In this paper, we saw how trivial it is to implement a k-NN classification algorithm with kdb+. Using tables and q-sql it can be implemented with three select statements at most, as shown in the util library at www.github.com/kxcontrib/knn. We also briefly saw how to use adverbs to optimize the classification time, and how data structures can influence performance.

Benchmarking this lazy implementation, with a random dataset available on the UCI website and using the Euclidean distance metric showed an average prediction accuracy of ~97.7%. While the classification time of a single instance after optimization of the code was 18ms per instance and the total validation time decreased significantly when using 4 slave threads (-s 4).

it's about time

Technical Whitepaper

2 An Introduction to Neural Networks with kdb+

Author:

James Neill works as a kdb+ consultant for one of the world's largest investment banks developing a range of applications. James has also been involved in the design of training courses in data science and machine learning as part of the First Derivatives training programme.

CONTENTS

2.1 INTRODUCTION

Due to the desire to understand the brain and mimic the way it works by creating machines that learn, neural networks have been studied with great interest for many decades. A simple mathematical model for the neuron was first presented to the world by Warren McCulloch and Walter Pitts in 1943.

With modern advances in technology and the computational power available, this field of research has had massive implications on how computers can be used to 'think' and learn to solve problems given an appropriate algorithm. A few interesting examples of this research in action include:

- Aiding in medical diagnostics
http://www.ijesit.com/Volume%202/Issue%202/IJESIT201302_33.pdf

- Interpreting art and painting images
http://googleresearch.blogspot.ca/2015/06/inceptionism-going-deeper-into-neural.html
http://arxiv.org/pdf/1508.06576v1.pdf

- Performing stock market predictions
http://reference.wolfram.com/applications/neuralnetworks/ApplicationExamples/12.2.0.html

A number of different algorithms have been developed around this field of research, and this paper is going to focus on the implementation of a feedforward neural network in kdb+. A feedforward network (also known as a multi-layer perceptron) is a type of supervised machine learning algorithm which uses a series of nonlinear functions layered together with an output. It can be used for classification or regression purposes and has been shown to be a universal approximator – an algorithm that can model any smooth function given enough hidden units[1].

This design of feedforward networks can be represented through operations on matrices and vectors. Array programming languages such as q/kdb+ are well suited for computational implementations in this format due to the vectorised operations on lists.

All tests were run using kdb+ version 3.2 (2015.05.07)

[1] Kurt Hornik, "Approximation Capabilities of Multilayer Feedforward Networks", *Neural Networks*, Vol. 4, pp. 251-257, 1991

2.2 FEEDFORWARD NETWORKS

As mentioned in the introduction, a feedforward neural network is a type of supervised machine learning algorithm. This means that it is trained on datasets for which the output for given inputs is already known.

A neural network consists of a set of neurons connected together in some order. Each neuron receives inputs from other neurons, performs some action on these inputs and outputs a signal that becomes input for another connected neuron.

2.2.1 The Perceptron

A single neuron by itself is often referred to as a perceptron (Figure 1). The idea is to construct a predictor based on a linear combination of the inputs and a set of weights which control the impact an input has on the system. This linear combination (the dot product of the inputs and the weights) is then used as the input to a chosen threshold function which produces the desired output.

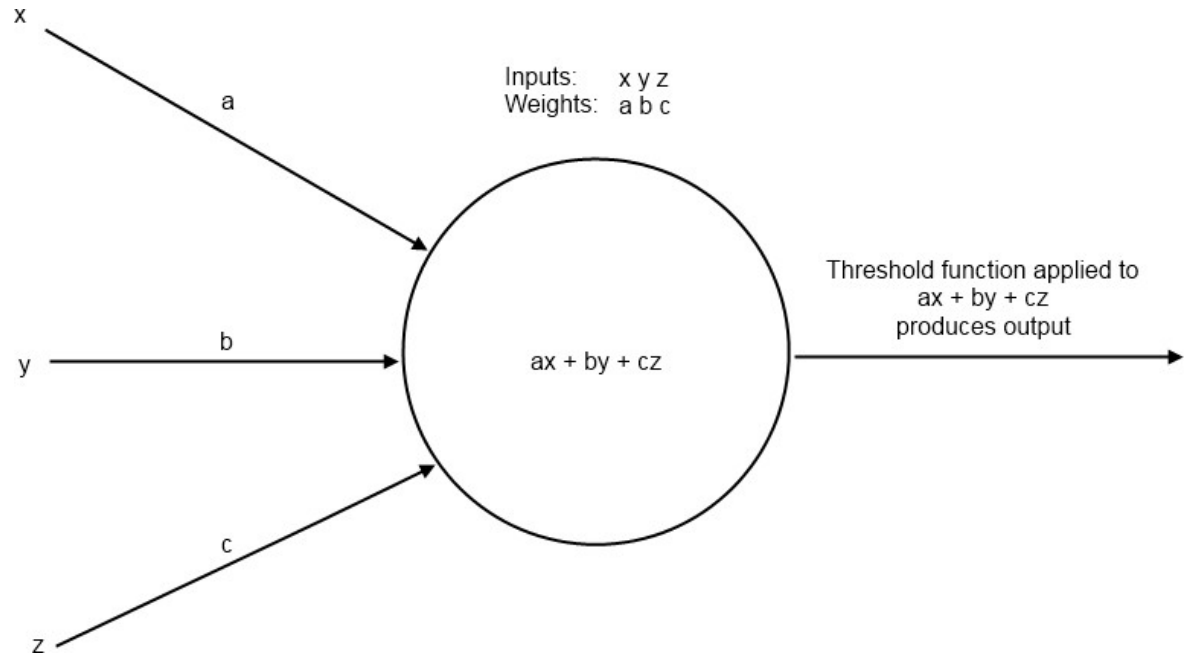


Figure 1: A Perceptron

2.2.2 Threshold Function

A threshold function represents the activation of the perceptron based on its inputs. A common choice for this function is the sigmoid function. This function provides a smooth curve bounded asymptotically by 0 and 1 on the vertical axis (Figure 2).

$$(x) = \frac{1}{1 + e^{-x}}$$

```
q)sigmoid:{1%1+exp neg x}
q)output:sigmoid[inputs mmu weights]
```

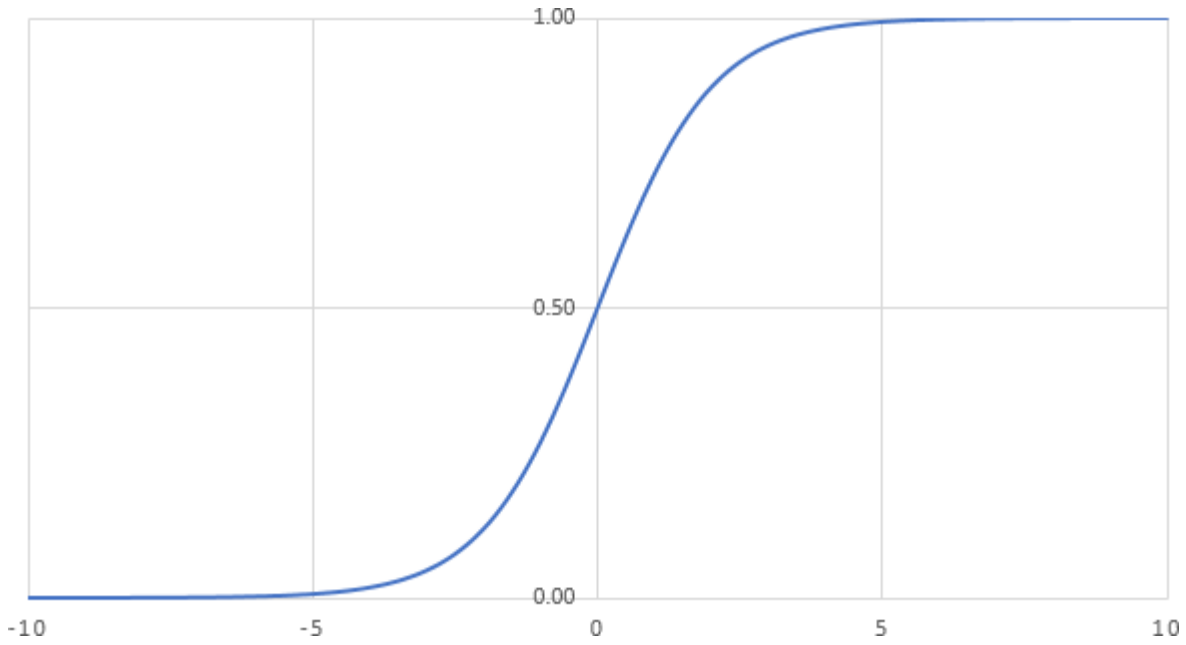


Figure 2: A plot of the sigmoid function

2.2.3 Perceptrons as Linear Predictors

Perceptrons can only solve linearly separable problems. A linearly separable problem is one where two sets in a plane can be cleanly divided by a straight line. We can demonstrate this by looking at plots of truth table outputs.

Let us say that a red dot represents true and a blue dot represents false. If we take the standard truth table inputs of 1 or 0 and add some random noise to them so that they are either slightly smaller than 1 or slightly larger than 0, then we get the plots shown in Figure 3.

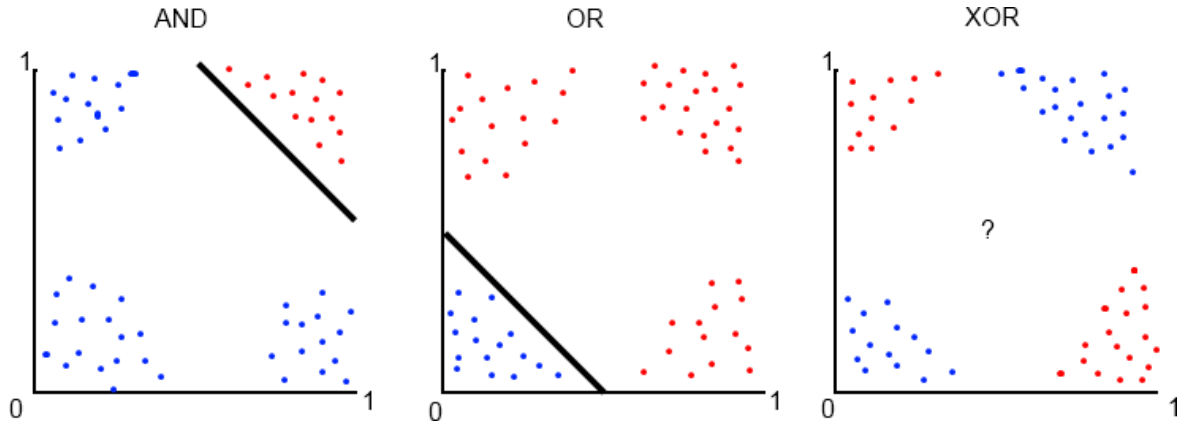


Figure 3: Truth Table Plots

Notice how we can easily separate the true results from false in the AND and OR truth tables with a straight line. Perceptrons are good predictors for these problems. However, if the data is not so easily separable, as with the XOR plot, we need a way to be able to make nonlinear predictions. Solving the XOR problem will form our motivation for connecting many perceptrons into a network.

2.2.4 A Neural Network

A network consists of neurons connected together in layers as shown in Figure 4. Each neuron in one layer is connected to each neuron in the next layer and each connection has an associated weight. The first layer of neurons is the input layer. The neurons in this layer represent input values from a data sample.

Following the input layer are the hidden layers. The number of neurons in a hidden layer is determined through a combination of background understanding of the problem and trial and error.

Finally there is the output layer. The number of neurons in this layer is determined by the type of problem to be solved. For example one might wish to classify an input data sample into one of two categories (e.g. true or false as in the XOR problem). This type of problem is referred to as a *binary classification* problem. For this type of problem we only need one output neuron because the sigmoid function will return values close to 1 or 0 after the network is trained.

However, the function which acts on the linear combination of inputs and weights at the output layer is not always the same as the threshold function used throughout the rest of the network. This is because we do not always desire a value between 0 and 1. Output functions for addressing different types of problems will be discussed in detail in Section 4.

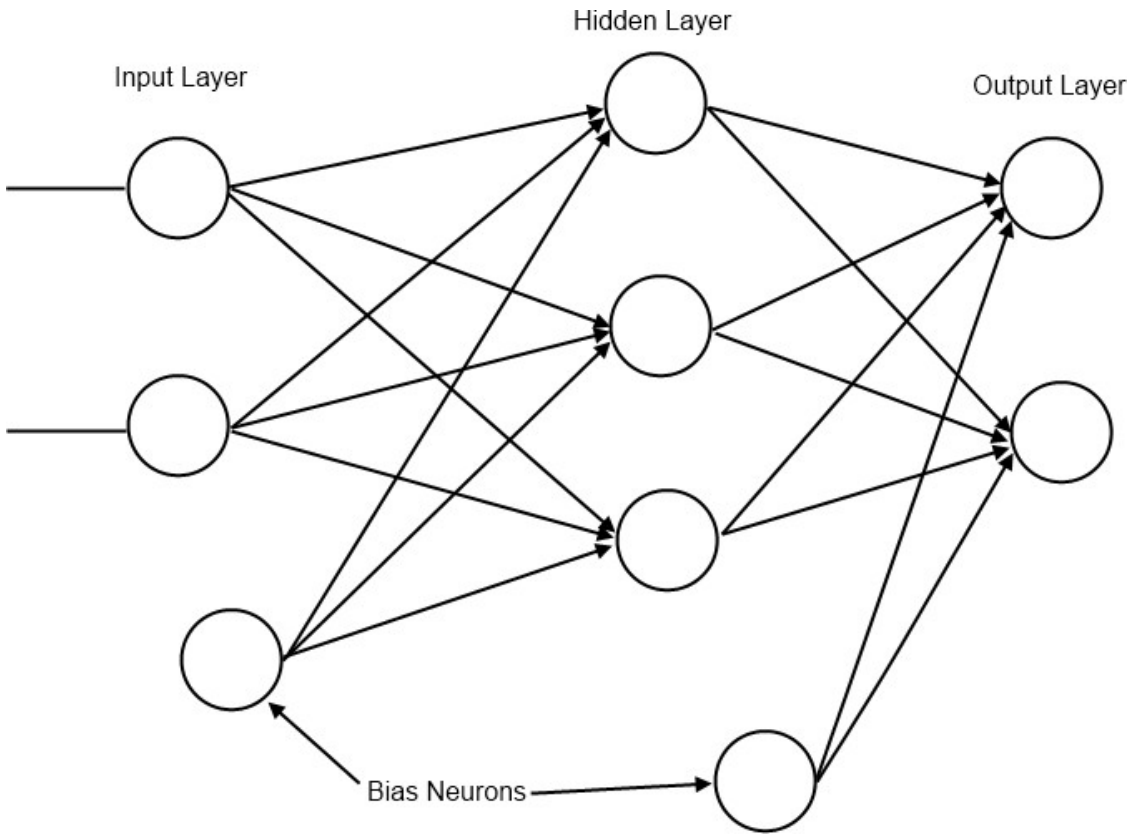


Figure 4: A feedforward network

2.2.5 Bias Neurons

A bias neuron is a neuron external to the main network. One is added to the input layer and one to each of the hidden layers. The value it passes to neurons in the next layer is always 1 and it receives no inputs from previous layers (see Figure 4). The purpose of bias neurons is analogous to the intercept parameter of a simple linear model – commonly written as $y = mx + c = \beta_1 x_1 + \beta_0 x_0$.

The absence of $\beta_0 x_0$ in the simple linear model results in the predicted line always passing through (0, 0) and the model will perform poorly when attempting to predict unknown values. Hence we always set x_0 to 1 and alter β_0 as we find the line of best fit. In the neural network we represent the network's version of the $\beta_0 x_0$ term as a bias neuron and associated weights. For more information on bias terms in statistical modelling see Chapter 3 in [2] and Chapter 7 in [1].

Bias neurons are added to the input layer by adding a 1 to each of the input samples in the data set used to train the network.

```
// Inputs and expected target values for XOR problem
q)input:((0 0f);(0 1f);(1 0f);(1 1f))
// Add a bias neuron to each input
q)input:input,'1.0
q)target:0 1 1 0f
```

2.2.6 Weight Initialization

The initialization process begins by assigning values to the weights present in the network. The weights are randomly assigned such that the values of the weights between the input nodes and a receiving node on the next layer are in the range (-1, 1) with mean 0.

Since there will be multiple neurons in the next layer we can represent all the weights between two layers as a matrix, where the number of rows represents the number of inputs and the number of columns represents the number of neurons in the next layer. An example of how this weight matrix and the input matrix interact is shown in Figure 5.

```
wInit:{
 // If only one input neuron is detected exit
 // This is most likely due to a missing bias neuron
 if[1=x;:"Number of input neurons must be greater than 1."];
 flip flip[r]-avg r:{[x;y]x?1.0}[y]each til x
}
// Init weights between 3 inputs and 4 outputs
// The first column represents the weights (connections) between
the 3 inputs and the first neuron in the next layer.
// The second column is the weights leading to the second neuron
in the next layer and so on.
q)wInit[3;4]
-0.3586151 0.09051553 -0.2815408 -0.05282783
0.02154042 0.4219367  0.2320934  -0.05853578
0.3370747  -0.5124522 0.04944742 0.1113636
```

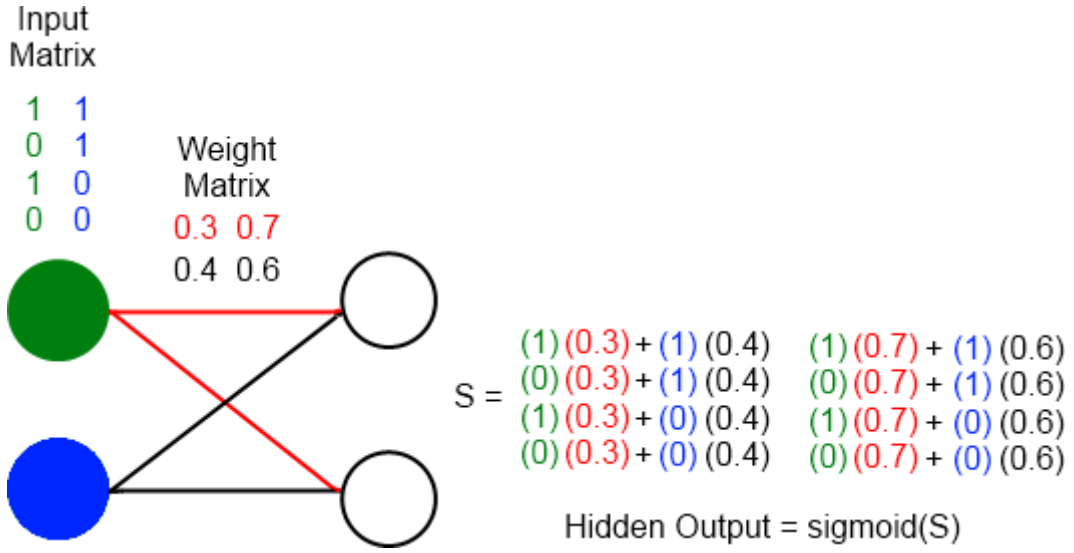


Figure 5: Diagram showing 2 input neurons (green and blue neurons) connecting to 2 hidden neurons. The colours in the matrices correspond with the area of the network those values are found during execution of a forward pass.

2.2.7 The Feedforward Network in kdb+

Once we have prepared the input data and the weights they can be applied to the network to provide output. We will use the network to predict the outputs of the XOR function.

```
 // weights between input layer and hidden layer (2 inputs + 1
bias neuron)
 q)w:wInit[3;4]
 // weights between hidden layer and output layer (4 hidden
neurons + 1 bias neuron)
 q)v:wInit[5;1]
 q)ffn:{[input;w;v]
   // Apply inputs and their weights to the hidden layer
   z:sigmoid[input mmu w];
   // Use output from hidden layer to generate an output
   sigmoid[z mmu v]
  }
 q)ffn[input;w;v]
 0.5028818
 0.5136649
 0.4891303
 0.5
```

The network has produced an output, but these values are not close to the target values. This is understandable as the weights have been randomly initialized. In order to produce the desired output the network must learn a more appropriate set of weights.

2.3 TRAINING THE NETWORK

Training a network to produce accurate results involves determining the weights which minimise the errors of the output nodes. The problem of minimising the error by adjusting the weights is solved by a technique called back-propagation - a form of gradient descent.

Gradient descent begins by calculating the derivative of the error function with respect to the weights. This derivative gives information about the direction needed to move along the surface described by the error function in order to arrive at a minimum. The weights are gradually adjusted using this information until an acceptable minimum in error has been reached.

2.3.1 Back-propagation

Back-propagation trains the network by propagating information about the error at the output back through the network and adjusting all the weights of the connecting neurons. For an output node that applies the sigmoid function the error function is the cross-entropy error function defined as:

$$-\sum_t y^t \log\hat{y} + (1 - y^t)\log(1 - \hat{y})$$

This gives us the following update rule for adjusting the weights between the output node and the hidden layer[2]:

$$\Delta v_h = \sum_t z_h^t (y^t - \hat{y})$$
$$v_h \leftarrow v_h + \alpha \Delta v_h$$

Where z_h^t is the output after evaluating the hidden neuron h for input sample t, v_h is the weight between the output neuron and hidden neuron h, y^t is the of target for sample t, $\hat{y}$ is the calculated output for sample t and α is the rate at which we adjust the weights (usually < 0.1).

Once the change in the above weights has been calculated we propagate the error to the hidden layer and generate the update rule for the weights between the input layer and the hidden layer:

$$\Delta w_h = \sum_t (y^t - \hat{y})\, v_h z_h^t (1 - z_h^t) x_j^t$$
$$w_{hj} \leftarrow w_{hj} + \alpha \Delta w_{hj}$$

Where w_{hj} is the weight between hidden neuron h and input neuron j and x_j^t is the input from neuron j for some sample t.

Using these formulas we can update our feed forward network function to implement back-propagation and allow training:

[2] The derivation of the update rules for back-propagation is beyond the scope of this paper. See Chapter 11 in [3] and Chapter 11 in [2].

```
// inputs - the input data set with bias node
// targets – known outputs corresponding to inputs
// lr – learning rate ‘alpha’
// d – dictionary with 3 items: output
//     weights between input and hidden layers
//     weights between hidden and output layers
q)ffn:{[inputs;targets;lr;d]
   z:1.0,/:sigmoid[inputs mmu d`w];
   o:sigmoid[z mmu d`v];
   // Error of output neurons
   deltaO:(targets-o);
   // Error of hidden neurons
   deltaZ:1_/:$[deltaO;flip d`v]*z*1-z;
   `o`v`w!(o;d[`v]+lr*flip[z] mmu deltaO;
           d[`w]+lr*flip[inputs] mmu deltaZ)
 }
```

```
// Example – the XOR problem
q)inputs
0 0 1
0 1 1
1 0 1
1 1 1
q)targets
0 1 1 0f
q)w
0.3257579   0.348099    -0.4320058 0.3356597
-0.07237444 -0.3028193  0.3088185  -0.3069554
-0.2533834  -0.04527963 0.1231874  -0.02870423
q)v
-0.0133154
0.04739764
0.2894549
-0.3235371
// Two training passes shows little change
q)finalResult:(ffn[inputs;targets;0.1]/)[2;`o`w`v!(0,();w;v)]
q)finalResult`o
0.5025557
0.5133001
0.4888265
0.4996545
// 10000 training passes shows significant improvement
q)finalResult:(ffn[inputs;targets;0.1]/)[10000;`o`w`v!(0,();w;v)]
q)finalResult`o
0.009305227
0.9890354
0.9890087
0.01142469
q)\ts finalResult:(ffn[inputs;targets;0.1]/)[10000;`o`w`v!(0,();w;v)]
164 2992
```

Now that the network has been trained it can be applied to a random permutation of XOR inputs to see how it performs on a single pass.

```
// generate a random permutation of list 0 to 99
q)rp:-100?til 100
// use rp to shuffle inputs and targets
// we generate rp first so that the shuffled indices of
// the inputs and targets match up
q)inputs:(100#inputs)rp
q)targets:(100#targets)rp
// using weights of trained network solve test inputs
q)rw:finalResult`w; rv:finalResult`v
q)res:(ffn[inputs;targets;0.1]/)[1;`o`w`v!(0;rw;rv)]
// Are all the predictions correct?
q) all raze[`int$res`o]=targets
1b
```

2.4 OUTPUT FUNCTIONS FOR REGRESSION AND MULTI-CLASS CLASSIFICATION

There are three types of output we are looking for when using a network of this type. The first we have already discussed in Section 2.4 – binary classification. We have also worked through an example of this problem when applying the network to predict XOR outputs. The other two problems we will discuss are multiclass outputs and nonlinear regression outputs.

2.4.1 Multiclass Outputs

For multiclass outputs the goal is to determine the correct classification of an input into three or more possible classifications. Unlike the binary classification situation, in the output layer there will be one neuron for each possible classification. The target values are transformed using one-hot encoding. This gives us a unique list of 0s and 1s for each possible classification that is the same length as the number of possible classifications. For example, if there are classifications A, B and C the transformations are 0 0 1, 0 1 0 and 1 0 0 – giving the output layer target patterns to match for training and testing. The output function used is the softmax function:

$$\hat{y}_i^t = \frac{\exp S_i^t}{\sum_k exp\ S_k^t}$$

Where $\hat{y}_i^t$ is the output from neuron *i* for sample *t*, S_i^t is the linear combination of outputs from the hidden layer and the weights connecting the hidden layer to output neuron *i* for sample *t* and S_k^t is the linear combination of outputs from the hidden layer and the weights connecting the hidden layer to output neuron *k* for sample *t*.

By using the softmax function we ensure that the sum of the outputs from each of the neurons in the output layer is 1. That allows us to pick the neuron with the highest output value as the most likely to be the classification we expect for the given input; the 'winning' neuron will be assigned a value of 1 and the other neurons a value of 0 resulting in a match to one of the one-hot encoded classifications. The cross-entropy error function in this case is:

$$-\sum_t \sum_i y_i^t \log \hat{y}_i^t$$

Where y_i^t is the target value for output neuron *i* with sample *t*.

The update rules are:

$$\Delta v_{ih} = \sum_t (y_i^t - \hat{y}_i^t) z_h^t$$

$$\Delta w_{hj} = \sum_t \left[\sum_i (y_i^t - \hat{y}_i^t) v_{ih}\right] z_h^t (1 - z_h^t) x_j^t$$

Where v_{ih} is the weight between output neuron *i* and hidden neuron *h*.

An example implementation of the softmax output will be shown in Section 5.

```
// one-hot encoding function
q)oneHot:{x!`float$x=/:x}
q)oneHot `A`B`C
A| 1 0 0
B| 0 1 0
C| 0 0 1
q)smax:{exp[x]%sum flip exp x}
q)smaxErr:{neg sum sum flip x*log y}
```

2.4.2 Nonlinear Regression Outputs

If the goal is to predict a real value, not necessarily constrained within the boundaries imposed by a threshold function, the output function is just the linear combination of the outputs from the hidden layer.

$$\hat{y}^t = \mathbf{v} \cdot \mathbf{z}^t$$

Where $\mathbf{v}$ is the vector of weights between the hidden layer and the output layer and $\mathbf{z}^t$ is the vector of outputs from the hidden layer.

In this case we change the error function from cross-entropy to the sum-of-squared errors:

$$\frac{1}{2}\sum_t (y^t - \hat{y}^t)^2$$

The update rules for a regression output are:

$$\Delta v_h = \sum_t (y^t - \hat{y}^t) z^t{}_h$$

$$\Delta w_h = \sum_t (y^t - \hat{y}^t) v_h z^t{}_h (1 - z^t{}_h) x^t{}_j$$

```
q)lin:{x}
q)linErr:{0.5*sum sum a*a:x-y}
```

It's useful now to put the different functions for error and output in dictionary format as this will allow us to use the same `ffn` function for all 3 types of classification:

```
// x is linear combination of hidden outputs and weights
q)outputFuncs:`sig`smax`lin!
    ({1%1+exp neg x};{exp[x]%sum flip exp x};{x})
// x is target value, y is calculated
q) errFuncs:`sig`smax`lin!
    ({neg sum sum flip(x*log y)+(1-x)*log 1-y};
     {neg sum sum flip x*log y};{sum sum a*a:x-y})
```

```
q)ffn:{[inputs;targets;lr;of;d]
 // Calculate the outputs of the hidden layer
 // and add bias node
 z:1.0,/:sigmoid[inputs mmu d`w];
 o:outputFuncs[of][z mmu d`v];
 // Error of output neurons
 deltaO:(targets-o);
 // Error of hidden neurons
 // Hidden bias node is not connected to any
 // input layer nodes so we drop it
 deltaZ:1_/:$[deltaO;flip d`v]*z*1-z;
 `o`v`w`err!(o; d[`v]+lr*flip[z] mmu deltaO;
             d[`w]+lr*flip[inputs] mmu deltaZ;
             errFuncs[of][targets;o])
}
```

2.5 CLASSIFICATION FOR 3+ CLASSES

As an example, we will study a set of Iris flower data which was originally introduced into research by Ronald Fisher in 1936. It contains samples from three different species of the Iris flower and has become a standard test case for many classification techniques within machine learning. By taking measurements of certain metrics (eg. length and width of sepals) the plants can be classified and computationally distinguished from each other. The data and a description of the data can be found in the links at http://archive.ics.uci.edu/ml/machine-learning-databases/iris.

We one-hot encode the different possible species of Iris, resulting in a neural network with 5 inputs (including the bias neuron), 7 hidden neurons (including the bias neuron) and 3 outputs. The data set is randomly shuffled to reduce the likelihood of a biased output. From this randomised selection of the data a random selection of 20 samples is taken as the test set and the other 130 samples are used in training.

```
// one-hot encoding of classes
q)IrisOneH:oneHot[distinct Iris.species]
q)IrisOneH
Iris-setosa    | 1 0 0
Iris-versicolor| 0 1 0
Iris-virginica | 0 0 1
q)Iris1h
slength swidth plength pwidth species     onehot
-----------------------------------------------
5.1     3.5    1.4     0.2    Iris-setosa 1 0 0
4.9     3      1.4     0.2    Iris-setosa 1 0 0
4.7     3.2    1.3     0.2    Iris-setosa 1 0 0
..
// Random permutation of the dataset
q)IrisRP:Iris1h{neg[x]?til x}count Iris1h
// Pick a test set – samples from data not used in training
q)IrisTest:IrisRP[-20?count IrisRP]
q)IrisTrain:IrisRP except IrisTest
// Init weights, input and output variables
q)w:wInit[5;6]
q)v:wInit[7;3]
q)input:1.0,'flip flip[IrisTrain]`slength`swidth`plength`pwidth
q)output:IrisTrain.onehot
// Train network
q)resIris:(ffn[input;output;0.01;`smax]/)[800;`o`w`v`err!(0;w;v;1f)]
// After 800 iterations (or epochs) how well does it perform?
q)all(IrisOneH?"f"$"j"$resIris`o)=IrisOneH?output
0b
q)100*sum[(IrisOneH?"f"$"j"$resIris`o)=IrisOneH?output]%count output
96.89922
// Init variables for test data
q)tinput:1.0,'flip flip[IrisTest]`slength`swidth`plength`pwidth
q)toutput:IrisTest.onehot
// Run test data through network without training
q)resIrisT:(ffn[tinput;toutput;0.01;`smax]/)[1;
  `o`w`v`err!(0;resIris`w;resIris`v;1f)]
q)all (IrisOneH?"f"$"j"$resIrisT`o)=IrisOneH?toutput
1b
```

Plot of error vs training epoch is given in Figure 6. We see that the error settles into an oscillation pattern around training epoch 700. While these oscillations are slowly converging it is likely that overfitting begins to take place shortly after the 700th iteration of training.

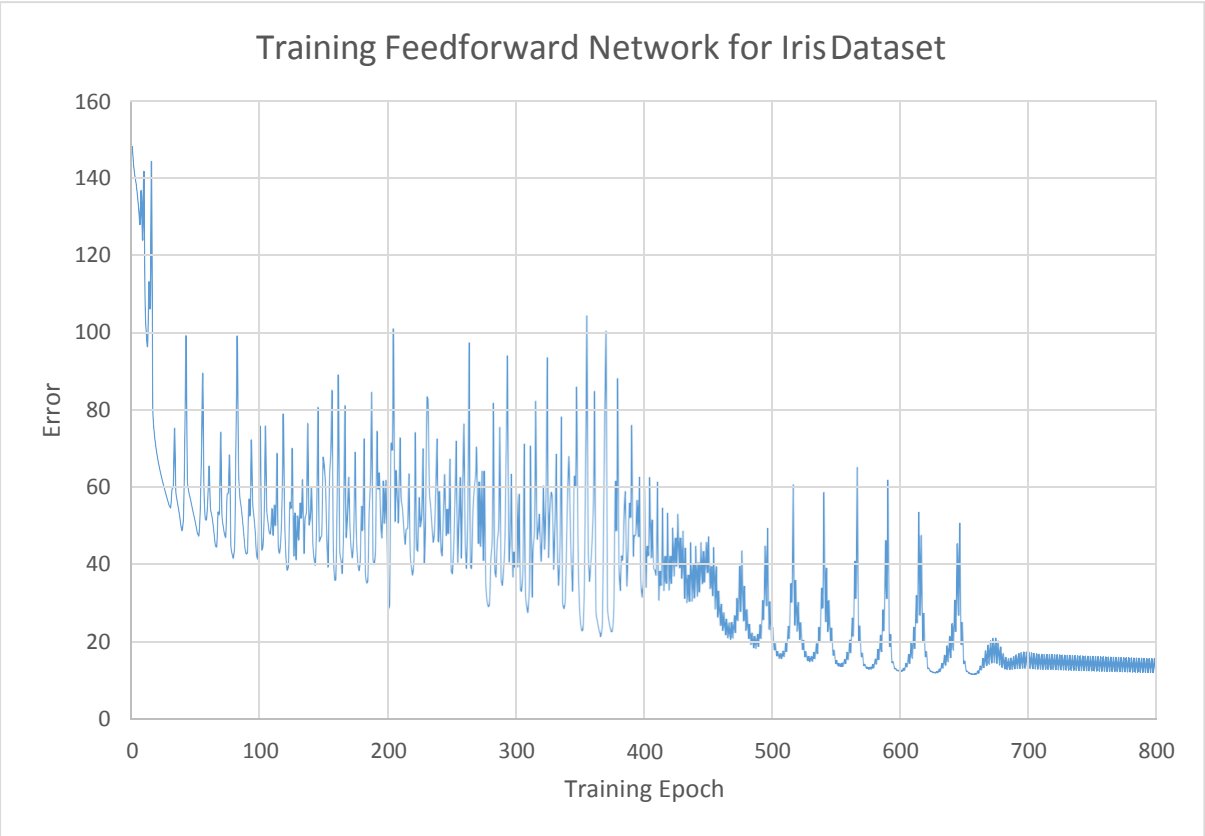


Figure 6: Error while training network on Iris dataset

2.6 STOCHASTIC GRADIENT DESCENT

Often, in practice, computing the error and gradient for the entire training set can be very slow and in some cases not possible if the data will not fit entirely in memory. Stochastic gradient descent solves this problem by performing the back-propagation training on small batches of the training set chosen at random.

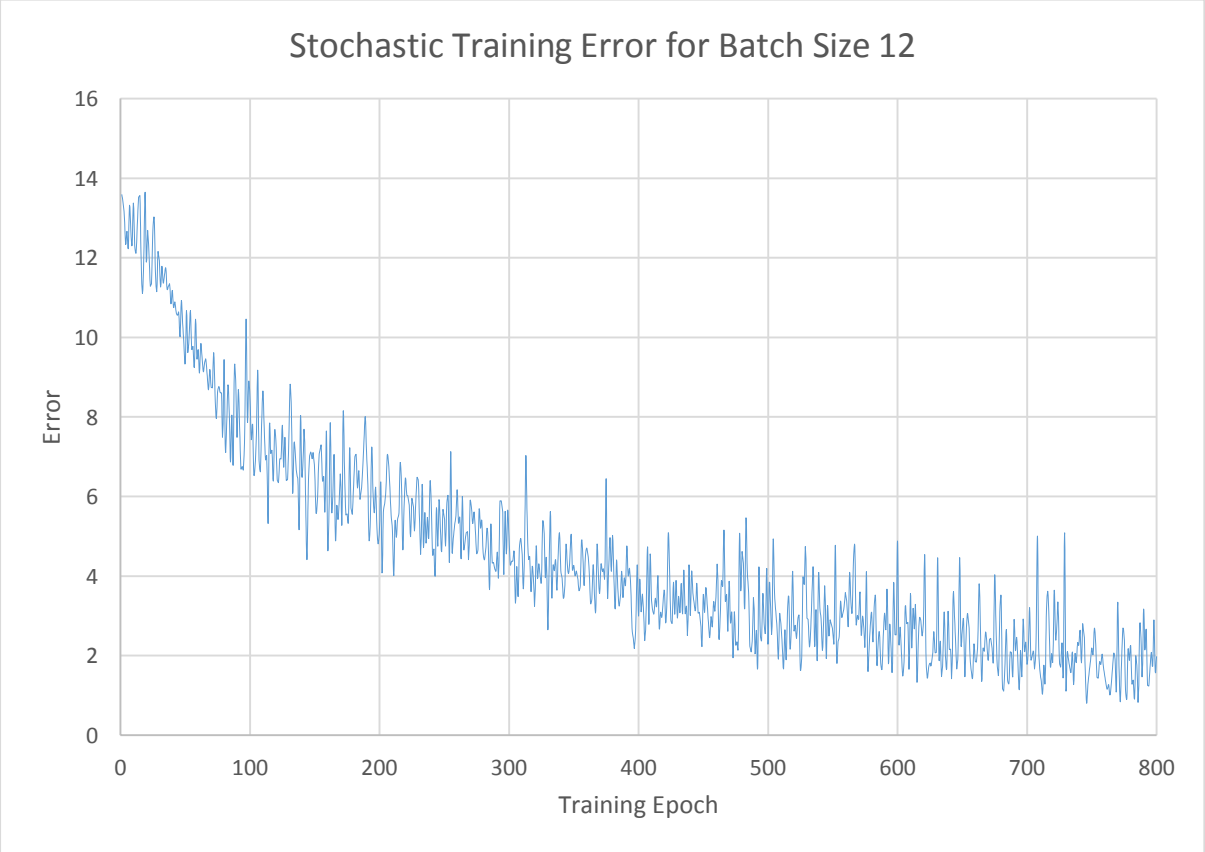
Randomly chosen batches simulate real-time streaming of data and help to reduce bias in the final trained network (see chapter 7 in [2] for more details on bias and variance in machine learning).

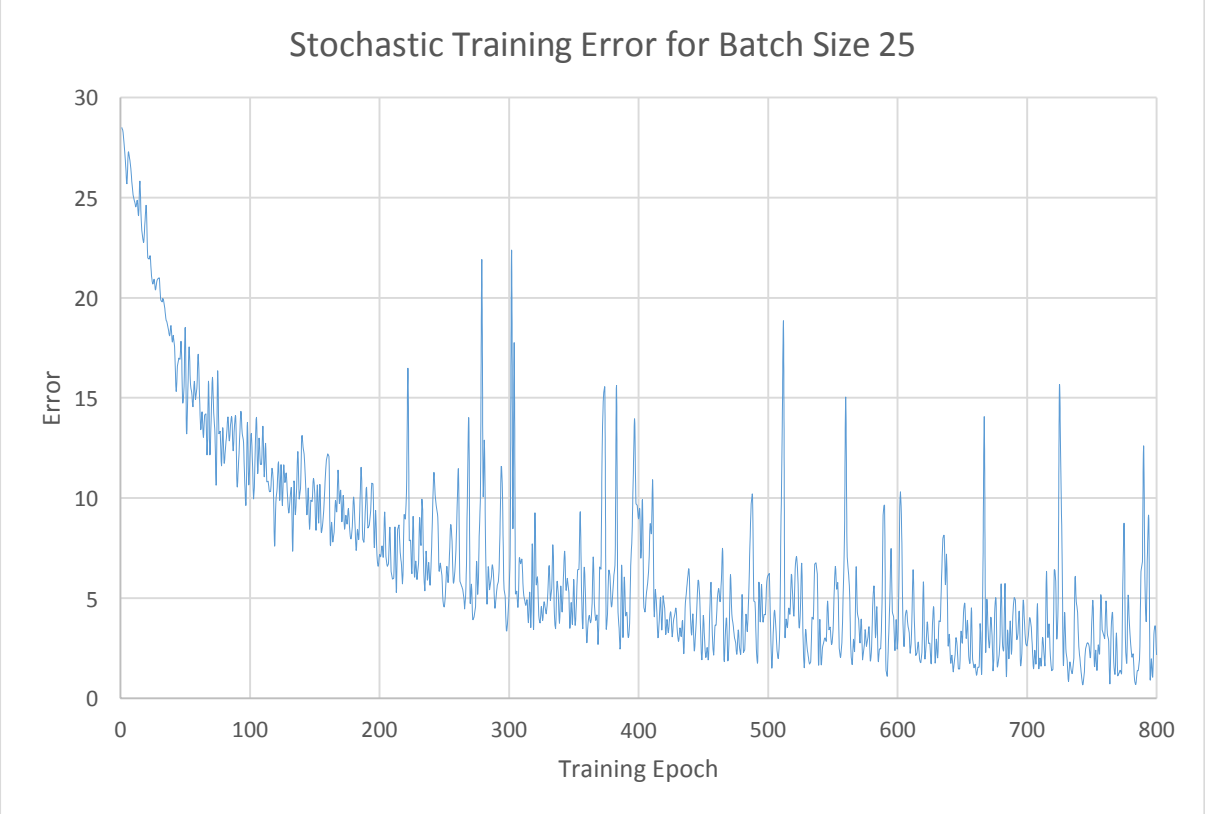
```
sffn:{[inputs;targets;lr;of;bsize;d]
 // random permutation for selection from inputs of size bsize
 rp:bsize?til count inputs;
 ffn[inputs[rp];targets[rp];lr;of;d]
}

// inputs[rp] and targets[rp] assume the input and targets
// are all contained in memory. If taking the data from
// disk a different query would be required to get the random
// selection of data.
// The important thing is that the input-target pairs in a
// batch are selected randomly with replacement (i.e. the
// same input-target pair can appear more than once in a
// batch)
```

2.6.1 Classification for 3+ classes using stochastic batch training

This time we will again use Fisher's Iris dataset, but the network will be trained using randomly selected small batches from the training data.

```
q)IrisTest:Iris1h[-20?count Iris1h]
q)IrisTrain:Iris1h except IrisTest
q)input:1.0,'flip flip[IrisTrain]`slength`swidth`plength`pwidth
q)target:IrisTrain`onehot
// choose a batch size of 12
q)resIris:(sffn[input;target;0.01;`smax;12]/)[800;
     `o`w`v`err!(0;w;v;1f)]
q)tinput:1.0,'flip flip[IrisTest]`slength`swidth`plength`pwidth
q)toutput:IrisTest.onehot
q) resIrisT:(ffn[tinput;toutput;0.01;`smax]/)[1;
    `o`w`v`err!(0;resIris`w;resIris`v;1f)]
q)all (IrisOneH?"f"$"j"$resIrisT`o)=IrisOneH?toutput
1b
```


Stochastic Training Error for Batch Size 12
Error
Training Epoch
0
2
4
6
8
10
12
14
16
0
100
200
300
400
500
600
700
800


Stochastic Training Error for Batch Size 25
Error
Training Epoch
0
5
10
15
20
25
30
0
100
200
300
400
500
600
700
800

2.7 CONCLUSION

In this whitepaper we have explored a proof-of-concept implementation of a feedforward network in kdb+. By constructing the model for the network using linear algebra (inputs, outputs and weights represented by matrices) we have shown that an array processing language is well suited to developing a complex system.

We have shown that multiple types of output functions can be easily applied to the network depending on the desired result. This generalisation demonstrates the flexibility of kdb+ to many problems where numerical data can be arranged into lists, arrays or tables.

In the event that there is not enough main memory to carry out the calculations on all the training data in one pass we presented an alternative approach, stochastic gradient descent, which allows the network to be trained on small batches of the data.

The network exhibited in this paper forms the groundwork for more complicated networks. Adding additional hidden layers and different options for threshold function will allow more complex convolutional and deep networks to be developed.

All tests were run using kdb+ version 3.2 (2015.05.07)

References:

[1] Murphy, K. P. (2012). *Machine Learning A Probabilistic Perspective.* MIT Press.

[2] Hastie, T., Tibshirani, R. and Friedman, J. *The Elements of Statistical Learning.* Springer, New York. (link to online version: http://statweb.stanford.edu/~tibs/ElemStatLearn/)

[3] Alpaydin, E. *Introduction to Machine Learning, Second Edition.* MIT Press.

Technical Whitepaper

3 Compression in kdb+

Author:

Eoin Killeen is based in New York. Eoin has worked as a kdb+ consultant on the design and development of a wide range of high-performance trading and analytics applications. He is currently working on a global real-time electronic trading visualization platform and an equity derivatives analytics portal at a US investment bank.

CONTENTS

3.1 INTRODUCTION

As the rate of data generation in financial markets continues to increase, there is a strong impetus to investigate how large data volumes can be more efficiently processed. Even if disk is considered cheap, it can be a valuable exercise for many applications to determine what improvements can be gleaned from an analysis of the use of compression. Aside from reduced disk costs, some use cases can gain significant performance improvements through a problem-specific approach to compression. For example, some systems have fast CPUs, but find that disk i/o is a bottleneck. In some such cases, utilizing CPU power to reduce the amount of data being sent to disk can improve overall performance.

Prior to kdb+ version 2.7, compression was achieved via file systems with inbuilt compression, such as ZFS. Version 2.7 introduced built-in OS-agnostic compression, which allowed on-disk data to be converted to compressed format using a range of algorithms and compression levels. This was expanded upon in version 2.8, with the addition of the ability to stream in-memory data directly to compressed format on disk. ZFS compression is still useful for some kdb+ applications, as it keeps cached data available for multiple processes. However, this paper will focus on the inbuilt data compression options provided by kdb+, which are available on all supported architectures.

Each system will have its own characteristics which determine the appropriate compression configurations to use. This paper will cover an introduction to compression in kdb+, a discussion of contributing factors to compression ratios and performance, and an analysis of how the use of compressed data can affect performance of some sample use cases.

All tests were run using kdb+ version 3.1 (2013.09.05)

3.2 COMPRESSION OPTIONS

There are two high-level approaches to saving on-disk data in compressed format. The first, available since kdb+ 2.7, is a two-step approach: save data to disk in the regular uncompressed format using set, then convert it to a compressed format using the -19! command. The second approach, available since kdb+ 2.8, is to stream data directly from memory to compressed format on disk by modifying the left- hand-side argument to set.

The first approach is useful for archiving existing historical data, or in cases where it is significantly faster to save the data to disk uncompressed, without the overhead of first compressing the data. In many other cases, it can be more convenient and/or performant to compress the data on the fly while saving down.

3.2.1 Converting saved data to compressed format using -19!

The specification for -19! from code.kx.com is:

```
-19!
(`:sourceFile;`:targetFile;logicalBlockSize;compressionAlgorithm;compr
essionLevel)
```

where

- logicalBlockSize is a power of 2 between 12 and 20 (pageSize or allocation granularity to 1MB - pageSize for AMD64 is 4kB, sparc is 8kB. Windows seems to have a default allocation granularity of 64kB). Note that this argument affects both compression speed and compression ratio (larger blocks can be slower and better compressed)

- compressionAlgorithm is one of the following, 0 - none, 1 - kdb+ ipc, 2 - gzip

- compressionLevel is between 0 and 9 (valid only for gzip, use 0 for other algorithms)

The compression algorithm and compression level inputs are self-explanatory. The logical block size determines the amount of data which will be compressed at a time in each block. A larger block will afford more opportunity for identification of repeated values, and hence a higher overall compression ratio. But this parameter also determines the minimum amount of data which can be decompressed at a time. If it is large relative to the amount of data that is likely to be accessed at a time, then a lot of unnecessary decompression work may be carried out during queries. Its lower bound is the system's allocation granularity, because dividing the data into smaller chunks than that would result in wasted space.

The various combinations of inputs will be discussed further in the following sections, but sample commands to save data to disk and then convert it to compressed format would look like this:

```
`:/db/trade_uncompressed set trade
-19! (`:/db/trade_uncompressed; `:/db/trade_compressed; 16; 1; 0)
```

Note that if this approach is used to compress data, it is preferable to have the source and target files on separate physical disks. This will reduce the number of disk seeks required to move the data iteratively in chunks. For migrating uncompressed databases to compressed format, Kx have provided a convenient script which can be found at http://code.kx.com/wsvn/code/contrib/simon/compress/cutil.q.

3.2.2 Saving in-memory data directly to compressed format on disk

In many cases, it can preferable to save data to disk in compressed format in a single step. This is more convenient than having a separate process to convert uncompressed on-disk data to compressed format after it has been saved. It can also be faster than the two-step method, depending on the compression algorithm and level used, and on system CPU and disk characteristics. Direct streaming compression has been implemented by overriding the left-hand-side argument to set:

```
(`:targetFile;logicalBlockSize;compressionAlgorithm;compressionLevel)
set table
```

3.2.3 Field-by-field compression

It is also possible to apply compression on a field-by-field basis, i.e. to use different compression rules from one table or column to another. Files of various compression levels (including no compression at all) can happily coexist inside a single table. So if some columns do not compress well or their usage patterns mean they don't benefit from compression, they can have different compression algorithms applied or be left uncompressed. This can be achieved by modifying the left argument to set to include a dictionary, mapping field names to the compression parameters to be used for each one. The null symbol in the key of this dictionary defines the default behavior, which will be applied to any fields not specified explicitly.

```
(`:splay/;``a`b!((17;2;9);(17;2;6);(17;2;6)))set([]a:asc
1000000?10;b:asc 1000000?10;c:asc 1000000?10)
```

3.2.4 Compression defaults

Rather than specifying the compression parameters individually every time set is called, we also have the option of defining default compression parameters which will be used if set is called the old-fashioned way, i.e. `` `:filename set table ``. This is done by defining the "zip defaults" variable in the dotz namespace, `.z.zd`. The format is the same as the non-filename parameters which are passed to `set`:

```
.z.zd:(17;2;6);`:zfile set asc 10000?`3
```

3.3 READING COMPRESSED DATA

The process of decompression is automatic and transparent to the user. Data consumers can read compressed files in the same way that they would read uncompressed ones, e.g. using get or by memory-mapping and querying the files. However, it is important to allow for some resource consumption associated with decompression when querying compressed data. While querying memory- mapped compressed data, any file blocks used by `select` will be decompressed into memory and cached there for the duration of the query. This is to avoid having to decompress the data twice – once during constraint evaluation, and once when being used as a selected column in the result set. Only required data will be decompressed into memory, but kdb+ will allocate enough memory to decompress the entirety of each vector touched by a query on compressed data. This means that memory use shown in `top` may indicate a higher number than expected, but only the required sub-set of this amount will actually be used. Increasing system swap space may be required, even if swap itself will not be used.

For frequently-repeated queries or regular usage patterns where the data is usually present in the OS cache, compressing data may be counterproductive. The OS can only cache the compressed data, and the decompressed queried data will only be cached by kdb+ for the duration of a query. So the decompression overhead must be overcome on every query. Because of this, columns which need to be accessed very frequently may perform better if left uncompressed.

It should be noted that because of the convenience in terms of automatic decompression, OS-level tools such as scp should be used if compressed data just needs to be moved about on disk or between hosts, without being manipulated in any way. This will prevent needless de/re-compression of the data, which would occur if q processes and IPC were used for this purpose.

3.4 WHAT DETERMINES THE COMPRESSION RATIO?

The compression ratio (uncompressed size vs. compressed size) is influenced by a number of factors. The choice of logical block size, compression algorithm and compression level will have a big impact here. But the compression ratio of any combination of inputs will also be heavily influenced by the nature of the data which is being compressed.

Some data-driven factors that influence the compression ratio are:

3.4.1 Number of distinct values vs. the total count of the vector

A low number of distinct values in a large vector could, for example, be grouped and referenced by index in each compressed block. A vector of entirely unique values offers less potential for efficient storage.

Similarly, sparsely populated columns will compress well, as the nulls can be compressed like a highly repeating value would be.

3.4.2 Level of contiguity of repeated values

A contiguous array of identical values can be described by the distinct value and the number of times it occurs. In some cases we can deliberately increase the level of contiguity by sorting tables on multiple columns before compressing. In the following example, we will compare the compression ratios achieved on a table of two highly-repeating columns – first unsorted, then sorted by one column, then sorted by both columns.

```
n:100000000
t:([]sym:n?`ibm`goog`aapl`tsla`spx;venue:n?`nsdq`nyse)
`:uncompressed/ set .Q.en[`:uncompressed] t
(`:unsorted/;16;2;5) set .Q.en[`:unsorted] t
(`:symSorted/;16;2;5) set .Q.en[`:symSorted] `sym xasc t
(`:symVenueSorted/;16;2;5) set .Q.en[`:symVenueSorted] `sym`venue xasc
t
```

Because the data is so highly repeating, we get a nice compression ratio of 10 even if the table is unsorted. But sorting on one column yields a significant improvement, and sorting on both means the entire 100m rows of these two fields can be persisted using a little over 1kb of disk. Obviously this sorting only increases the compression ratios for the columns that are actually sorted – other columns won't see improvements.

Table Version	Compression Ratio
Unsorted	10.19
Sorted by sym	30.47
Sorted by sym, venue	669.55

3.4.3 Datatype of the vector

If a vector of high-precision datatype such as long contains a high proportion of values which could have been expressed as ints or shorts, some of the unused precision can be "reclaimed" in a way through compression. Booleans also compress particularly well, but this is a special case of the above point regarding repeating values, as there are only two possible distinct values in a Boolean column.

The below test table contains Boolean, int and long columns. The int and long columns contain the same values. Taking a sample compression configuration of gzip with logical block size 16 and compression level 5, we can see how each data type compresses:

```
n:100000000
ints:n?0Wi
t:([]Boolean:n?01b;integer:ints;longint:`long$ints)
`:nocomp/ set t
(`:comp/;16;2;5) set t
```

The compression ratios achieved on each field show that Booleans have compressed by a large factor, and the longs have also compressed significantly. A lot of the storage cost of using long precision has been reclaimed.

Field	Compression Ratio
Boolean	5.97
Int	1
Long Int	1.62

In some cases, the extra precision afforded by longs is required only for a small proportion of values. In such cases, compression allows us the benefit of increased precision where it is needed, while staying close to the disk cost of regular ints where it is not.

3.5 EFFECTS OF COMPRESSION ON QUERY PERFORMANCE

The user-specifiable parameters for compression provide a lot of room for experimentation and customization, according to the demands of a particular system. For a given use case, the choice of compression parameters should be informed by a number of factors:

- CPU time taken to perform the compression
- change in disk i/o time
- compression ratio achieved
- usage patterns for the on-disk data

The relative weighting of these factors will vary from one system to another. If reduction in physical disk usage is the primary motivating concern behind the use of compression, then naturally the compression ratio itself must be maximized, while constraining the other parameters to acceptable levels.

The focus of our tests will be on the impact to downstream users of the data rather than the impact on the writing process. Random access to compressed data is supported, which means only required blocks of the file will be decompressed. The choice of logical block size will determine the minimum amount of data which can be decompressed at a time, so if logical block size is large relative to the average amount of data retrieved at a time, then redundant work may be performed in decompressing the data. If the gzip algorithm is used, then the tradeoff between compression level and time taken to (de/)compress must also be considered.

3.5.1 Setup

Each system will have its own performance characteristics, and experimentation with the compression parameters should be done on a case-by-case basis. We will measure the impact of using compressed rather than uncompressed data across a range of query types. The tests will be run first with the data present in the OS cache, and then with the cache completely cold, having been emptied before each test using the cache flush functionality of the io.q script provided by Kx at http://code.kx.com/wsvn/code/contrib/simon/io/io.q. Each test will be performed 10 times (with the cache being fully flushed before each iteration of the cold cache tests) and timings will be averaged. For hot cache results, the first query time will not be included in the average. This is to allow for initial caching from disk.

All tests are performed using kdb+ 3.1 2013.09.05 on a Linux Intel(R) Xeon(R) L5520 16 CPU/4 core 2.27GHz with 144GB RAM. Disk used is ext3 SAN.

We will use a basic test schema of a trade and quote table, with 100m rows of trades and 400m rows of quotes, evenly distributed throughout a trading day.

```
n:100000000
st:.z.D+09:30
et:.z.D+16:00
trade:([]sym:asc n?`3; time:"p"$st+((et-st)%n-1)*til n; price:n?1000.;
size:n?100)
n*:4
quote:([]sym:asc n?`3; time:"p"$st+((et-st)%n-1)*til n; bp:n?1000.;
ap:n?1000.; bs:n?100; as:n?100)
```

3.5.2 Procedure

The above test tables were saved to disk, both uncompressed and compressed using the kdb+ IPC algorithm, with low, mid- and high-range logical block sizes.

The relative changes in performance (compressed vs. uncompressed) of the following query types were measured:

- Select all data from the table
- Select all data for a subset of symbols
- Aggregate values by symbol
- Aggregate values by symbol and time bucket
- Asof join straight from disk

3.5.3 Results

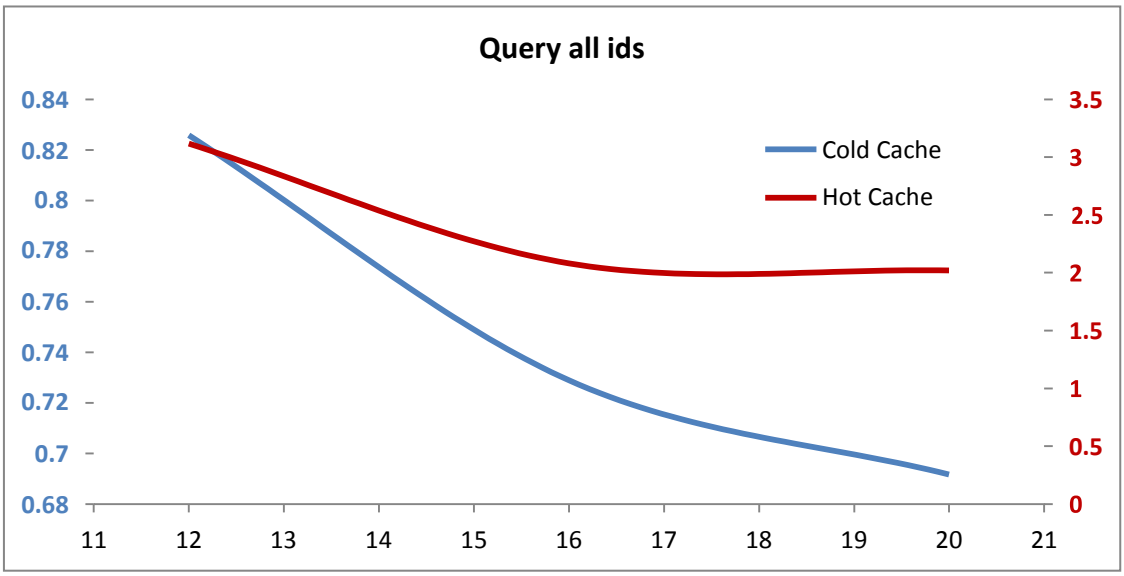
The following charts plot the ratio of performance of queries on compressed data vs. the performance of the same query on uncompressed data. So a value of 2 means the query on compressed data took twice as long as its uncompressed counterpart, and a value less than 1 means the performance improved on compressed data.

The compression ratios achieved did not change significantly across logical block sizes:

Logical Block Size	Compression Ratio
12	1.5
16	1.55
20	1.55

3.5.3.1 Select all data from the table

Query: select from trade where i<>0



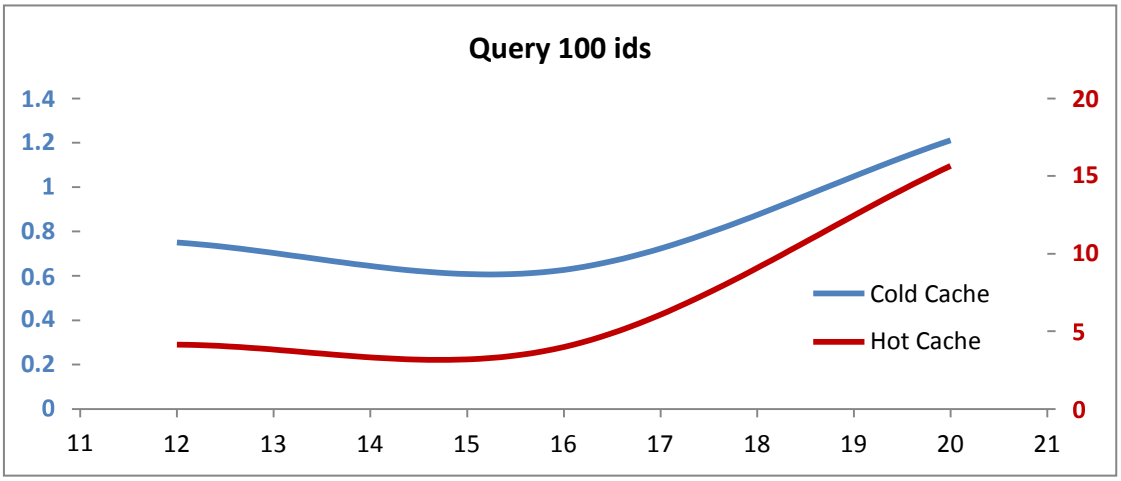
When querying the entire table with cold OS cache, we see significant improvements in performance using compressed data. This is because there is a reduction in the amount of raw data which must be read from disk, and the cost of decompression is low when compared to this benefit. The relative improvement increases with logical block size, as decompressing an entire vector can be done more efficiently when there are fewer blocks to read and coalesce.

In the case of hot OS cache, we see that performance has degraded to 2-3 times its original level. This illustrates the cost of the OS having access only to compressed raw data. When the cache is hot, the decompression cost is high relative to the total time taken to get the uncompressed data into memory.

3.5.3.2 Select all data for a subset of symbols

Query: select from trade where sym in ids

Ids is a vector of 100 distinct randomly selected symbols from the trade table, which was created prior to cache flushing.

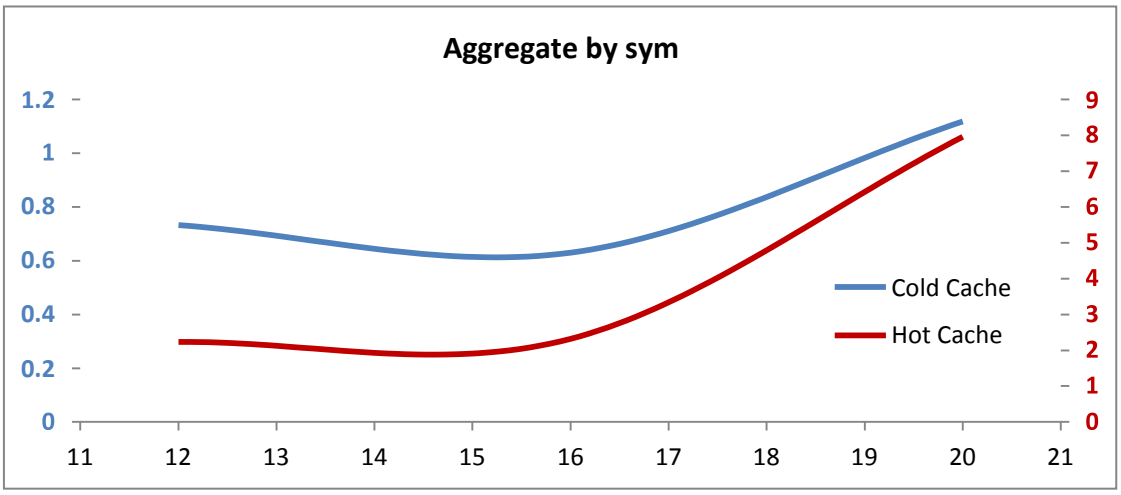


Querying on 100 symbols with a cold cache, we see moderate improvements in query performance on compressed data at low logical block sizes. With larger logical blocks, we need to carry out some redundant decompression work, as we only require sub-sections of each block. The cost of this redundant work eventually cancels out the disk read improvement for queries on this particular number of symbols.

With a hot cache, querying on a subset of symbols gets a lot slower with large logical blocks. This illustrates the benefit of fully random access (i.e. no redundant work), which is afforded by using uncompressed data. However, these queries still return in the range of hundreds of milliseconds, so a significant increase in query time may still be worth it, depending on the compression and performance desired in a particular case.

3.5.3.3 Aggregate values by a subset of symbols

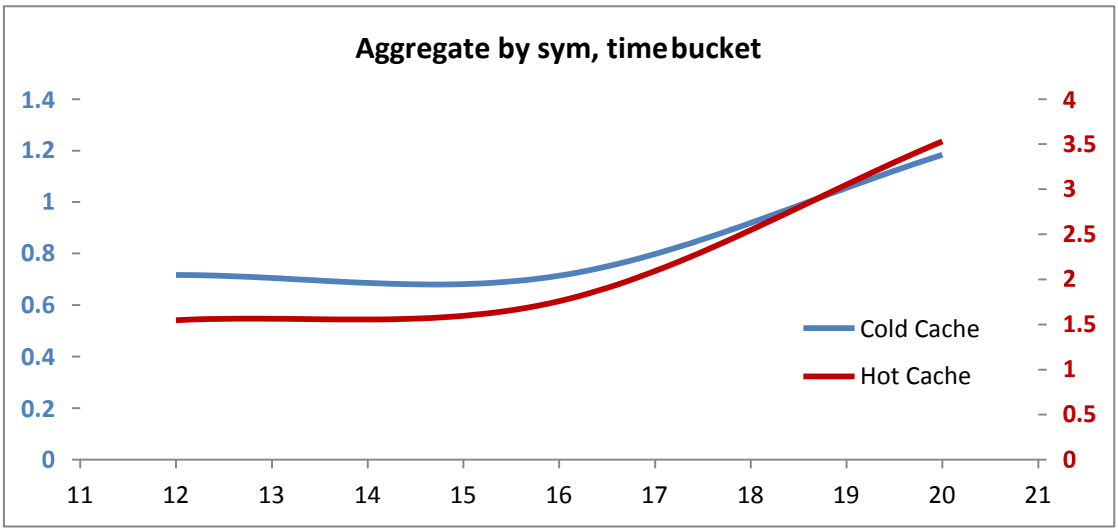
Query: select size wavg price by sym from trade where sym in ids



When we add the additional work of aggregating data by symbol, we don't see much change in the case of a cold cache. But with a hot cache, the increased CPU load of aggregation reduces the relative advantage of compressed data. Data read time is now slightly less of a contributing factor towards overall performance.

3.5.3.4 Aggregate values by a subset of symbols using time buckets

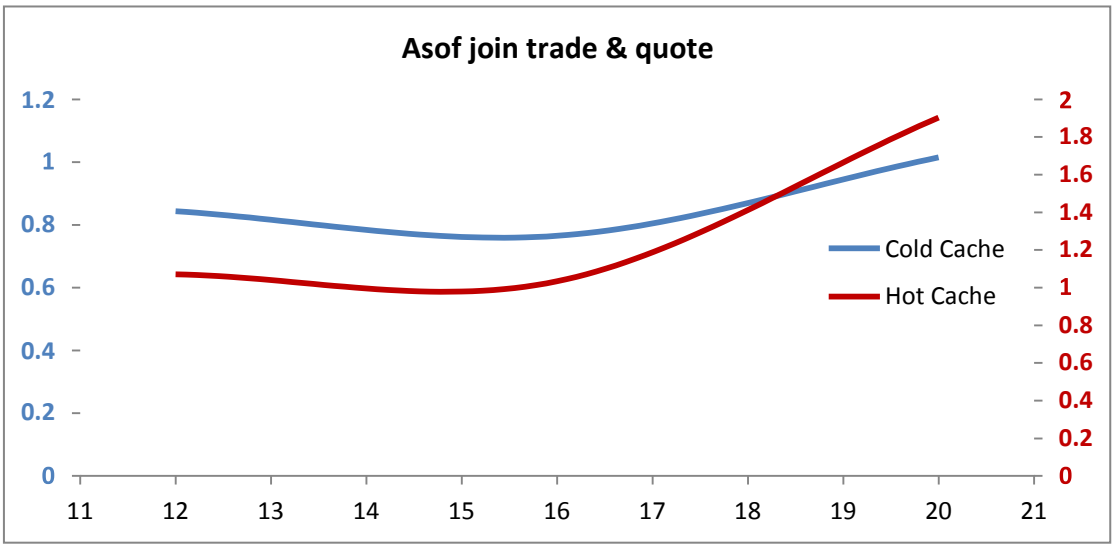
Query: select size wavg price by sym,5 xbar time.minute from trade where sym in ids



Aggregating by time bucket in addition to by sym, we see that the relative decrease in performance on compressed data has again reduced, as data retrieval time is less of a factor.

3.5.3.5 Asof join a trade and quote table

Query: aj[`sym`time;select from trade where sym in ids;select from quote]



The final test is a highly CPU-intensive asof join of the trade and quote records for 100 symbols. Extending the pattern of previous tests, we can see that performance against compressed trades and quotes is quite comparable to results against uncompressed tables.

3.6 CONCLUSION

In any given application, the metrics which will determine if and how compression should be used will be informed by the individual use case. Our tests have shown that for this particular setup, assuming some OS caching, the more CPU-intensive queries will see significantly less overall impact than simple reads on small, randomly distributed pieces of data. This implies that we could expect to see acceptable performance levels for intensive queries, while achieving worthwhile reductions in disk usage, i.e. our compression ratio of 1.5. Experimenting with logical block sizes suggested that we should keep this parameter at the low end of the spectrum for our queries, or we will see a falloff in performance. This is re-enforced by the relative lack of change in compression ratio across logical block sizes. Depending on the individual use case, other applications may choose to optimize for overall compression ratio or time taken to save down the data.

it's about time

Kdb+ and WebSockets

4 **Kdb+ and Websockets**

Author:

Chris Scott works a kdb+ consultant at one of the world's largest financial institutions developing a range of kdb+ applications which use WebSockets as a form of communication. Chris has also been involved in designing HTML5 training courses and building HTML5 mobile and desktop applications for First Derivatives' Kx technology.

CONTENTS

4.1 INTRODUCTION

Since the release of kdb+ 3.0, it has been possible to make use of WebSockets when connecting to a kdb+ process. This has big implications on how we can build front-end applications that use a kdb+ back-end, in particular for applications wanting to display real-time information.

WebSockets are supported by most modern browsers, and so for the first time it is possible to hook a web application up directly to a kdb+ process to allow real-time communication over a persistent connection. This gives us the ability to build a pure HTML5 real-time web application which can connect to kdb+ through JavaScript, which is native to web browsers.

Using web applications to build GUIs for kdb+ applications is becoming increasingly popular. Only a web browser is required to run the GUI so it is OS independent. This means that when developing the GUI, only one codebase is required for all platforms and maintenance and upgrades are much easier and efficient to perform.

This whitepaper will introduce what WebSockets are and what benefits they hold over standard HTTP. It will also take the reader through the set-up of a simple web page that uses WebSockets to connect to a kdb+ process and the steps involved in passing data through the connection.

We will look at different methods for converting data between kdb+ and JavaScript and ways to help ensure the connection is secure.

The paper will finish up by providing a full working example (including code) of a simple web application which uses a kdb+ to provide real-time updating tables based on user queries.

NOTE: As well as q, this paper will make significant use of HTML, CSS and JavaScript. A basic understanding of these will be necessary to take full advantage of this paper, though any complex structures will be examined here in detail. For a crash course on the above languages, please refer to the following resources:

- http://www.w3schools.com
- http://www.codecademy.com

Throughout the paper, kdb+ code will be in grey text boxes while HTML and JavaScript code will be in green text boxes.

4.2 WEBSOCKETS

4.2.1 What Are WebSockets

WebSockets are a long awaited evolution in client and web server communication technology. They provide a protocol between a client and server which runs over a persistent TCP connection. The client-server connection can be kept open as long as needed and can be closed by either the client or the server. This open connection allows bi-directional, full-duplex messages to be sent over the single TCP socket connection - the connection allows data transfer in both directions, and both client and server can send messages simultaneously. All messages sent across a WebSocket connection are asynchronous.

Without WebSockets, bi-directional messaging can be forced by having distinct HTTP calls for both the client sending requests and the server publishing the responses and updates. This requires either the client to keep a mapping between outgoing and incoming messages, or the use of a proxy server in between the client and server (known as HTTP tunneling). WebSockets simplify this communication by providing a single connection which both client and server can send messages across.

WebSockets were designed to be implemented in web browsers and web servers, but they can be used by any client or server application. The ability for bi-directional real-time functionality means it provides a basis for creating real-time applications on both web and mobile platforms. The WebSocket API and Protocol have both been standardised by W3C and the IEFT respectively.

4.2.2 Why Not Just Use .z.ph?

It has previously been possible to connect to a kdb+ process using HTTP. HTTP requests could be processed using the `.z.ph` and `.z.pp` handlers. To illustrate this, simply start up a q process with an open port and then type the `hostname:port` into your web browser. This will give a very basic view of the q process.

Straight out of the box, this is very simple, and provides a useful interface for viewing data and running basic queries without being limited to the character limits of a q console. If, however, you want to do more than just simple analysis on the q process, this method presents a few drawbacks:

1) Customisation is quite a complicated process that requires you to manipulate the functions in the `.h` namespace which form the basis of the in-built HTTP server code. The HTML markup is generated entirely by the q process.
2) Data must be requested by the browser, so some form of polling must occur in order to update the webpage. This makes the viewing of real-time data impossible without the continuous polling of the kdb+ server.
3) `.z.ph` uses synchronous messaging, and the webpage is effectively refreshed every time a query is sent.

Instead of relying on `.z.ph` to serve the entire web page, an application could make use of AJAX techniques to send asynchronous GET and POST requests to a kdb+ server. `.z.ph` and `.z.pp` can be modified to handle these requests. AJAX (Asynchronous JavaScript And XML) is a descriptive name that covers a range of web development techniques that can be used to provide asynchronous messaging between a web application and server.

This allows requests to be made by the client in the background without having to reload the web page, and as there is no reliance on `.z.ph` to generate the HTML markup, the application can be fully customised independent of the kdb+ process. In order to receive updates the application will still have to poll the server, which is not ideal for real-time data. The full implementation of this method is beyond the scope of this whitepaper.

As WebSocket connections are persistent, the server is able to push updates to the client rather than relying on the client to poll for the information. Additionally, while each HTTP request and response must include a TCP header, WebSockets only require this header to be sent during the initial handshake.

4.3 CONNECTING TO kdb+ USING WEBSOCKETS

4.3.1 The Handshake

In order to initialise a WebSocket connection, a WebSocket 'handshake' must be successfully made between the client and server processes. First, the client sends a HTTP request to the server to upgrade from the HTTP protocol to the WebSocket protocol:

```
// Client WebSocket request header
GET /text HTTP/1.1
Host: localhost:5001
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Key: xc0DPEL132mlMtGdbWJPGQ==
Sec-WebSocket-Version: 13
```

The server then sends a HTTP response indicating that the protocol upgrade was successful:

```
// Server WebSocket response header
HTTP/1.1 101 Switching Protocols
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Accept: 1thgMROs9ylOWOMkc2WUWGRzWdY=
```

4.3.2 .z.ws Message Handler

Like the `.z.ph` HTTP GET handler, kdb+ has a separate message handler for WebSockets called `.z.ws`, meaning all incoming WebSocket messages will be processed by this function. There is no default definition of `.z.ws`; it should be customised by the developer to handle the incoming messages as required. Later, several examples of a customised `.z.ws` handler will be shown, but initially we will look at a very basic definition:

```
q) .z.ws:{neg[.z.w].Q.s value x;}
```

For now, let's assume that `x` (the query from the client) is being passed in as a string. `value x` simply evaluates the query, and passing this into `.Q.s` will present the result in console output format (i.e. plain text) .

As mentioned before, all messages passed through a WebSocket connection are asynchronous. This means that we must handle the server response within the `.z.ws` message handler. `neg[.z.w]` does this by asynchronously pushing the results back to the handle which raised the request.

From the server side, using `neg[.z.w]` within the handler will push the result back to the client once it has been evaluated. The client does not wait for a response and instead the open WebSocket connection will receive the response some time later, so we must handle this on the client side as well.

Fortunately, JavaScript has a native WebSocket API which allows us to handle this relatively easily. The main parts of the JavaScript API are explained below.

```
<script>
function connect(){
  if ("WebSocket" in window){  // check if WebSockets supported
     // open a WebSocket
     var ws = new WebSocket("ws://host:port");
     ws.onopen = function(){
        // called upon successful WebSocket connection
     };
     ws.onmessage = function(msg){
        // called when the client receives a message
     };
     ws.onclose = function(){
        // called when WebSocket is closed.
     };
  }
  else{
     // the browser doesn't support WebSockets
  }
}
ws.send(msg)   // function to handle sending a message
</script>
```

4.3.3 Simple Example

Here is an example of a simple client application connecting to a kdb+ server using a WebSocket. This is a slightly modified version of the example that can be found on the code.kx.com WebSocket cookbook. http://code.kx.com/wiki/Cookbook/Websocket

First, start up our kdb+ server and set our `.z.ws` handler. Here we will add some error trapping to the handler to send the error message to the client.

```
C:\Users\Chris>q -p 5001  // Open a port on 5001
KDB+ 3.2 2014.09.21 Copyright (C) 1993-2014 Kx Systems
w32/ 2()core 3689MB Chris chris-hp 169.254.80.80 NONEXPIRE

q) .z.ws:{neg[.z.w].Q.s@[value;x;{`$x}];}
q)
```

Next we create a HTML document we will use to connect to the above kdb+ server. The below code can be copied into a text editor and saved as a .html file, then opened in any modern web browser.

```
<!doctype html>
<html>
<head>
      <title>WebSocket Demo</title>
</head>
<!-- when page loads, call connect function -->
<body onload="connect();">
  <!-- The application will consist of a text box to enter queries, a
button to execute the queries, and an area to display the results -->
  <textarea id="txtInput" placeholder="q)"></textarea>
  <!-- the onclick event handler will make a call to JavaScript -->
  <button id="cmdInput" onclick="send();">Go</button>
  <div id="txtOutput"></div>
  <script> // JavaScript code goes inside 'script' tags
  var ws, cmd = "";  // initialise the JS variables required
  var input=document.getElementById("txtInput");
  var output=document.getElementById("txtOutput");
  function connect(){
  // this is where we define the WebSocket API functionality
    if ("WebSocket" in window) {
      ws = new WebSocket("ws://localhost:5001/");
  // show the state of the WebSocket connection in the display area
      output.value="connecting...";
      ws.onopen=function(e){output.innerHTML="connected";}
      ws.onclose=function(e){output.innerHTML="disconnected";}
      ws.onmessage=function(e){
      // when a message is received, print the message to the display area
along with the input command
        output.innerHTML = cmd +
             e.data.replace(/ /g, ' ').replace(/\n/g, '<br />') +
output.innerHTML;
      // the message is in plain text, so we need to convert ' ' to
' ' and '\n' to '<br />' in order to display spaces and newlines
correctly
        cmd="";
      }
      ws.onerror=function(e){out.value=e.data;}
    } else alert("WebSockets not supported on your browser.");
  }
  function send(){
   // store the input command as tmpInput so that we can access it later
to print in with the response
    cmd = "q)" + input.value + "<br />";
   // send the input command across the WebSocket connection
    ws.send(input.value);
   // reset the input test box to empty, and focus the cursor back on it
ready for the next input
    input.value=""; input.focus();
   }
  </script>
  <style>  /* define some CSS styling on page elements */
  #txtInput {width: 85%; height: 60px; float:left; padding: 10px;}
  #cmdInput {margin-left: 10px; width:10%; height:80px; font-weight:
bold;}
```

```
  #txtOutput {width: 96%; height: 300px; font-family: courier new;
padding: 10px; border: 1px solid gray; margin-top: 10px; overflow:auto;}
  </style>
</body>
</html>
```

This example sets our webpage up as a q console. As it is using `.Q.s` to convert the kdb+ output into string format, it is subject to the restrictions of the q console size set by `\c`.

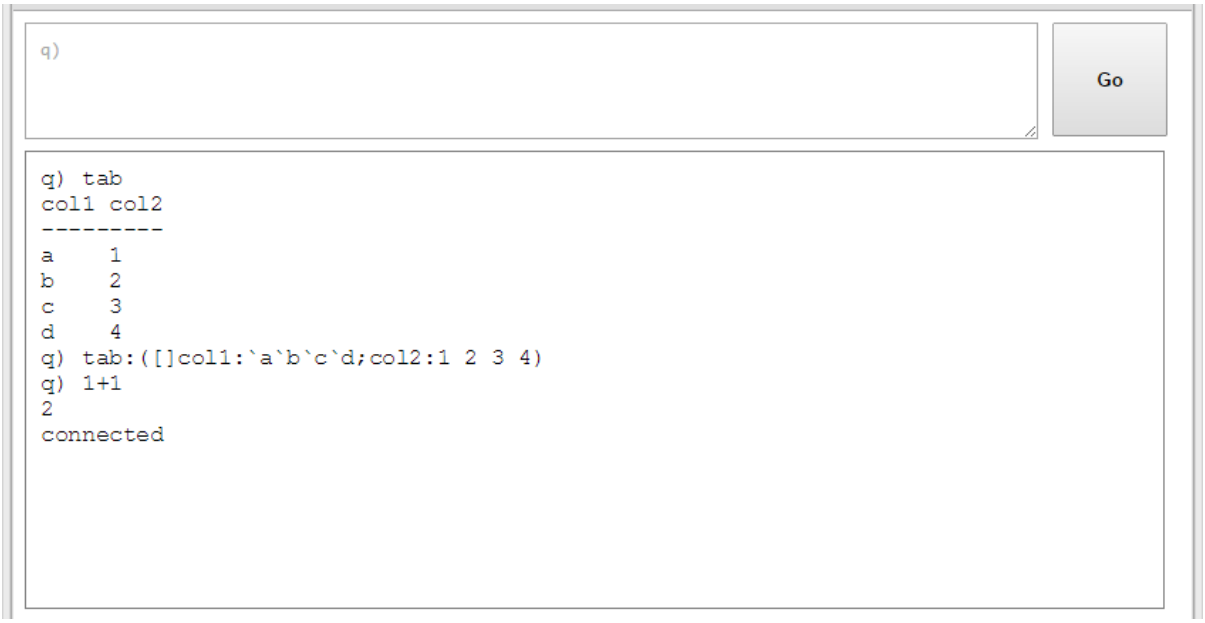


Figure 3.2 – This example provides a web console application which allows q commands to be executed from the browser

4.3.4 Pushing Data To The Client Using neg[h]

Above, we have used `neg[.z.w]` within the `.z.ws` handler to return the result to the client immediately after it is received and evaluated. But in some situations we don't want to just provide a response, but rather set up a subscription that continually pushes data through our WebSocket connection.

As the WebSocket is persistent and obeys normal IPC protocol, we can push data through it asynchronously at any time using `neg[h]`, where `h` is the WebSocket handle. To see this, in the input box of the example application above, type the following code and click 'Go'.

```
.z.ts:{[x;y]neg[x].Q.s .z.T}[.z.w]; system"t 1000"
```

You should now see the current time being output every second, without the browser having to poll for the data.

4.4 CONVERTING kdb+ TO JSON

Converting kdb+ into string format using `.Q.s` means that not only is the message limited by the console size, but also that as the client is receiving plain text the data will be unstructured and very difficult to parse into something that can be manipulated by the client.

JavaScript has a built-in technology called JSON (JavaScript Object Notation) which can be used to create JavaScript objects in memory to store data. These objects are collections of name/value pairs which can be easily accessed by JavaScript and used to build the data structures displayed on the web page.

We want to make use of JSON to hold our data so that we can easily display and manipulate our data on the client. There is a very important relationship between kdb+ and JSON – kdb+ dictionaries and JSON objects are comparable. This means that we can parse our kdb+ data into JSON structures very easily, with tables just becoming arrays of JSON objects.

Depending on the application, it may be of benefit to do this conversion on the server side within the q process, or on the client side in the JavaScript code. We will explore the two options along with an example to show both in action.

4.4.1 Using json.k Within a q Process (Server-side Parsing)

kdb+ 3.2 includes a new `.j` namespace which contains functions for translating between kdb+ and JSON format. For older versions of kdb+, Kx provides the same functionality inside a script named json.k which can simply be loaded into a q process. That script can be found here: http://kx.com/q/e/json.k

There are two main functions that we will make use of:
- **`.j.j`** parses kdb+ into a JSON string
- **`.j.k`** parses a JSON string into kdb+

```
C:\Users\Chris\Documents\Whitepaper>q
KDB+ 3.2 2014.09.21 Copyright (C) 1993-2014 Kx Systems
w32/ 2()core 3689MB Chris chris-hp 169.254.80.80 NONEXPIRE

q)tab:([]col1:`a`b`c`d;col2:1 2 3 4)
q)tab
col1 col2
---------
a    1
b    2
c    3
d    4
q).j.j tab
"[{\"col1\":\"a\",\"col2\":1},{\"col1\":\"b\",\"col2\":2},
{\"col1\":\"c\",\"col2\":3},{\"col1\":\"d\",\"col2\":4}]"
q).j.k "{\"a\":1,\"b\":2,\"c\":3,\"d\":4}"
a| 1
b| 2
c| 3
d| 4
```

Here is an example of how we can use this to manipulate data in our web app. First set up our server and use `.j.j` to convert the kdb+ output to JSON:

```
C:\Users\Chris>q -p 5001  // Open a port on 5001
KDB+ 3.1 2014.03.27 Copyright (C) 1993-2014 Kx Systems
w32/ 2()core 3689MB Chris chris-hp 169.254.80.80 NONEXPIRE

q) .z.ws:{neg[.z.w].j.j@[value;x;{`$"'",x}];}
q)
```

We can edit the `ws.onmessage` JavaScript function from the example in Section 2 to now handle the messages coming through as JSON strings rather than just plain text.

```
ws.onmessage=function(e){
      var t,d = JSON.parse(e.data);
// we will use the variable t to build our HTML
// we parse the JSON string into a JSON object using JSON.parse
      if (typeof d == "object") { // either a table or dictionary
            if (d.length) {  // if d has a length it is a table
// we will to iterate through the object wrapping it in the HTML table
tags
               t = '<table border="1"><tr>'
               for (var x in d[0]) {
// loop through the keys to create the table header
                  t += '<th>' + x + '</th>';
               }
               t += '</tr>';
               for (var i = 0; i < d.length; i++) {
// loop through the rows, putting tags around each column value
                  t += '<tr>';
                  for (var x in d[0]) {
                     t += '<td>' + d[i][x] + '</td>';
                  }
                  t += '</tr>';
               }
               t += '</table>';
            } else {
// if the object has no length, it must be a dictionary, we will iterate
over the keys to print the key|value pairs as would be displayed in a q
console
                  t = "";
                  for (var x in d) {
                        t += x + " | " + d[x] + "<br />";
                  }
            }
      } else {
// if not an object, then the message must have a simple data structure,
in which case we will just print it out.
            t = d;
      };
       output.innerHTML = cmd + t + "<br />" + output.innerHTML;
 }
```

This gives us the same web console application as before, but the distinction is that we are now handling JSON objects which are easier to manipulate than strings of plain text. The JavaScript code will also be aware of data type information which is not the case with plain strings.

Using the `.j` functions within the q process is very straight forward, and if it is suitable for the application to transfer messages across the WebSocket as strings this can be a good solution. However, in some cases it may be preferable to serialise the data into binary format before transmitting it across the connection.

4.4.2 Using c.js Within JavaScript (Client-side Parsing)

Instead of parsing the data within the kdb+ server, we could instead use `-8!` to serialise the data into kdb+ binary form and then deserialise it on the client side directly into a JSON object. With client machines generally being a lot more powerful than in the past, it is reasonable to provide the client side JavaScript code with some extra workload.

This approach requires a little more understanding of JavaScript. However, Kx provides the script c.js which contains the functionality to serialise and deserialise data on the client side. The `deserialize` function converts kdb+ binary data into JSON, while the `serialize` function will convert our message into kdb+ binary format before sending it to the server. c.js can be found here: http://kx.com/q/c/c.js,

Section 4.1 showed how we can parse q structures into JSON strings and send them to our client. In this example, we will instead do all of the parsing on the client side to produce the same result. The client will send strings to the server but the server will send serialised data back to the client. Outgoing messages are serialised using the `-8!` operator:

```
C:\Users\Chris>q -p 5001  // Open a port on 5001
KDB+ 3.2 2014.09.21 Copyright (C) 1993-2014 Kx Systems
w32/ 2()core 3689MB Chris chris-hp 169.254.80.80 NONEXPIRE

q).z.ws:{neg[.z.w] -8! @[value;x;{`$"'",x}];}
q)
```

We need to make a few adjustments to our JavaScript code from section 4.1 in order to handle the serialised responses. Firstly, we need to load the c.js script into our web page.

```
<script src="http://kx.com/q/c/c.js"></script>
```

The above line should be placed on the line before the opening `<script>` tags which surround the rest of the JavaScript code.

When we define our WebSocket connection, we need to tell the JavaScript code to expect the messages to be coming through in binary. For this we use `ws.binaryType = 'arraybuffer'`. Place this line on the line after we have set up the new WebSocket connection.

```
ws = new WebSocket("ws://localhost:5001/");
ws.binaryType = 'arraybuffer';
```

Next, we need to edit the `ws.onmessage` function. As we are deserialising the message straight into JSON, we do not need to use the `JSON.parse` function. Instead, we simply replace it with the `deserialize` function provided in c.js.

```
var t,d = deserialize(e.data);
```

The rest of `ws.onmessage` is identical to the example in Section 4.1.

More information on serialisation can be found here:
http://code.kx.com/wiki/Reference/ipcprotocol

4.5 WEBSOCKET SECURITY

Security is one of the biggest concerns in any system. It is of the utmost importance that, especially when dealing with sensitive financial information, users only have access to view and edit data they are allowed to see.

The methods used to protect kdb+ processes can be extended to cover WebSockets. There is one caveat in terms of username/password authentication which we will discuss below. A full discussion on kdb+ permissioning is beyond the scope of this whitepaper; instead, please refer to the Kx Technical Whitepaper on permissions on this link:
https://code.kx.com/q/wp/permissions_with_kdb.pdf

4.5.1 Username and Password

Client authentication is not defined in the WebSocket protocol, and in most cases it is up to the developer to implement an authentication method of their choice. This means that being prompted for a username and password by the browser when the WebSocket makes a request cannot be guaranteed. Firefox does support this, but Chrome and other browsers do not, as the following example will show.

First create our user/password file and start the server using the –u argument.

```
C:\Users\Chris>type users.txt
chris:password
user2:password2
C:\Users\Chris>q -p 5001 –u users.txt // Open a port on 5001
KDB+ 3.2 2014.09.21 Copyright (C) 1993-2014 Kx Systems
w32/ 2()core 3689MB Chris chris-hp 169.254.80.80 NONEXPIRE

q) .z.ws:{neg[.z.w] -8!@[value;-9!x;{`$"'",x}];}
```

If we try to connect to the process in Firefox it will prompt for a username and password and, upon correct entry, successfully create the connection. Chrome, on the other hand, will not ask for authentication details and so the connection will not be successful. When the connection is not successful due to authentication, a response code of 401 (unauthorized) will be returned to the client.

Behind the scenes, Firefox is actually sending two requests. When it receives the first 401 response code, it prompts for a username and password and then sends a new WebSocket request with an additional 'Authorization' option in the header. Chrome and other browsers do not currently do this for WebSocket connections.

As a workaround, we can make a standard HTTP request before we initalise the WebSocket. Once authorization has been granted to the session, it will continue to include the Authorization option in any subsequent HTTP headers that are sent. When we then try to initalise our WebSocket connection, the header will include this information and access will be granted.

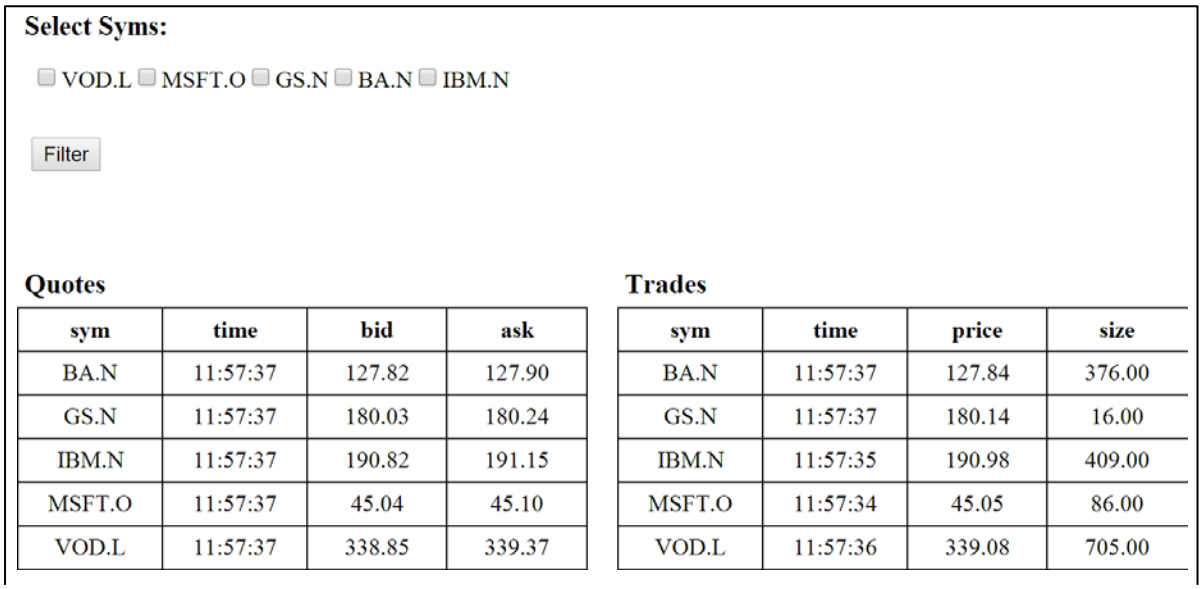
As mentioned in Section 3.2, all WebSocket messages will be processed by the `.z.ws` handler. To fully secure a kdb+ system, this handler should be locked down in the same manner as the `.z.pg`, `.z.ps`, `.z.ph` and `.z.pp` handlers.

NOTE: Stunnel can be used to secure any WebSocket server using the OpenSSL library. For more information on Stunnel, see:
http://code.kx.com/wiki/Cookbook/Websocket#Secure_sockets:_stunnel

4.6 A SIMPLE EXAMPLE – REAL-TIME DATA

This section will present a simple example in which some tables will be updated in the browser in real-time.

The full code can be found in Appendix A along with start-up instructions in order to get the example working. Some of the key features will be explained here. Once the example is successfully running you should be able to see the following tables in your browser continuously updating:



sym	time	bid	ask
BA.N	11:57:37	127.82	127.90
GS.N	11:57:37	180.03	180.24
IBM.N	11:57:37	190.82	191.15
MSFT.O	11:57:37	45.04	45.10
VOD.L	11:57:37	338.85	339.37

sym	time	price	size
BA.N	11:57:37	127.84	376.00
GS.N	11:57:37	180.14	16.00
IBM.N	11:57:35	190.98	409.00
MSFT.O	11:57:34	45.05	86.00
VOD.L	11:57:36	339.08	705.00

Figure 6.1 – The web page shows the last quote and trade values for each symbol, along with giving the user the ability to filter the syms in view

The idea behind the pubsub mechanism here is that a client will make subscriptions to specific functions and provide parameters that they should be executed with. Standard kdb+ IPC allows messages to be sent as a list, where the first element is a function name and the following elements are function parameters. We will mimic that functionality here. Messages will be sent as lists (arrays in JavaScript) in the format `(function;arg1;arg2;…)`.

```
.z.ws:{value -9!x}
```

In this example we will be serializing all messages that are sent from the client to the server. Accordingly, on the server we will need to use `-9!` to deserialise the message into kdb+ format.

Next we initialize the `trade` and `quote` tables and `upd` function to mimic a simple Real Time Subscriber, along with a further table called `subs` which we will use to keep track of subscriptions.

```
// subs table to keep track of current subscriptions
subs:2!flip `handle`func`params`curData!"is**"$\:()
```

The `subs` table will store the handle, function name and function parameters for each client. As we only want to send updates to a subscriber when something has changed, we store the current data held by each subscriber so that we can compare against it later.

```
// pubsub functions
sub:{`subs upsert (.z.w;x;y;res:eval(x;enlist y));(x;res)}
pub:{neg[x] -8!y}
pubsub:{pub[.z.w] eval(sub[x];enlist y)}
.z.pc:{delete from `subs where handle=x}
```

The `sub` function will handle new subscriptions by upserting the handle, function name and function parameters into the `subs` table. When a new subscription is made we execute the function with the specified parameters, store the result in the `subs` table along with the other subscription details, and then return a list of `(functionName; data)`.

When we publish data to a client, we want to send it asynchronously and serialized. This is handled by the `pub` function. `pubsub` combines these two functions such that the `subs` table is updated with the latest data and this data is then pushed to the client. Shortly we will see how the `subs` table can be used to continually push updates to the client.

Very importantly, if a connection is closed we want to make sure that there are no subscriptions left for that handle in the subs table. This is handled in `.z.pc`.

The functions that can be called and subscribed to by clients through the WebSocket should be defined as necessary. In this example, we have defined some simple functions to display the last record for each `sym` in both the `trade` and `quote` tables.

Now that we have defined the subscription and publication methods, we need a way of determining when data has changed and then pushing the latest information to the client. The `refresh` function will run on a timer and execute each function in the subs table with the specified parameters. If any of the function calls produce a different output to what the client holds (stored in the `curData` column) then the latest data will be pushed to that client.

```
refresh:{
  update curData:{[h;f;p;c]
    if[not c~d:eval(f;enlist p);pub[h] (f;d)]; d
  }'[handle;func;params;curData] from `subs
 }
```

One thing to note is that this example keeps the amount of data sent to the client very small, so there is no problem with storing a copy of what each client currently has for each subscription.

In the JavaScript code for the client, most of the functionality has been seen in previous examples. One thing to highlight is how we can use the messages as arrays to know which JavaScript function to call.

```
ws.onmessage=function(e){
   var d =  deserialize(e.data);
   window[d.shift()](d[0]);
};
```

Once the message in `e.data` has been deserialised, it will be in the format `['functionName',parameters]`. This keeps a certain symmetry between how the messages are handled on both client and server. In order to execute a function when we have the name as a string, we can use `window['functionName']`. `d.shift()` returns the first element of the array d, whilst also persisting d with the first element removed.

Inside the `tableBuilder` function, there is some formatting for times and numbers.

```
data[i][x].toLocaleTimeString().slice(0,-3)
data[i][x].toFixed(2)
```

The end result is a simplistic, interactive, real-time web application showing the latest trade and quote data for a range of symbols. Its intention is to help readers understand the basic concepts of kdb+ and WebSocket integration.

4.7 CONCLUSION

This whitepaper has shown how WebSockets can be used as part of an HTML5 GUI to connect to a q process to allow persistent, real-time communication between kdb+ and a web browser.

There are different methods to sending messages over the connection, and we have seen how to parse kdb+ data into JSON objects using both server-side and client-side methods. Both of these methods have their benefits and drawbacks so it is important to consider the application infrastructure when deciding which method will be most suitable. Serialising data across the connection is easy to achieve using the `-8!` and `-9!` functions on the kdb+ server and the c.js code provided by Kx on the JavaScript client.

Access control is a crucial aspect of any system, and we have seen that although basic user authentication is not part of the standard WebSocket protocol it can be overcome using techniques which are already employed in front-end applications using AJAX as a method of communication.

Delta Dashboards, part of the Delta Suite from First Derivatives, provides a range of great ways to visualise and analyse both kdb+ data and data from other sources while leveraging the enterprise functionality of Delta for authentication/authorisation. For more information on Delta Dashboards please visit here.

All tests were run using kdb+ version 3.2 (2014.09.21)

The release of kdb+ 3.2 contained updates to allow kdb+ processes to be used as a WebSocket client, and also for compression to be used over a WebSocket connection. For more information: http://code.kx.com/wiki/Cookbook/Websocket#Compression

APPENDIX A

Below are four separate scripts which can be copied into a text editor and saved with the respective names in the same directory. Start the q processes up first, and then open the HTML file in a web browser.

pubsub.q
Start this process first. It will create the q interface for the WebSocket connections and contains a simple pubsub mechanism to push data to clients when there are updates.

```
// q pubsub.q -p 5001

.z.ws:{value -9!x}

// table and upd definitions
trade:flip `time`sym`price`size!"nsfi"$\:()
quote:flip `time`sym`bid`ask!"nsff"$\:()
upd:insert

// subs table to keep track of current subscriptions
subs:2!flip `handle`func`params`curData!"is**"$\:()

// pubsub functions
sub:{`subs upsert (.z.w;x;y;res:eval(x;enlist y));(x;res)}
pub:{neg[x] -8!y}
pubsub:{pub[.z.w] eval(sub[x];enlist y)}
.z.pc: {delete from `subs where handle=x}

// functions to be called through WebSocket
loadPage:{pubsub[;`$x]each `getSyms`getQuotes`getTrades}
filterSyms:{pubsub[;`$x]each `getQuotes`getTrades}

// get data methods
getData:{
  w:$[all all null y;();enlist(in;`sym;enlist y)];
  0!?[x;w;enlist[`sym]!enlist`sym;()]
 }
getQuotes:{getData[`quote] x}
getTrades:{getData[`trade] x}
getSyms:{distinct (quote`sym),trade`sym}

// refresh function - publishes data if changes exist, and updates subs
refresh:{
  update curData:{[h;f;p;c]
    if[not c~d:eval(f;enlist p);pub[h] (f;d)]; d
  }'[handle;func;params;curData] from `subs
 }
// trigger refresh every 100ms
.z.ts:{refresh[]}
\t 100
```

fh.q

This will generate dummy trade and quote data and push it to the pubsub process. The script can be edited to change the number of symbols and frequency of updates.

```
// q fh.q
h:neg hopen `:localhost:5001 // connect to rdb
syms:`MSFT.O`IBM.N`GS.N`BA.N`VOD.L // stocks
prices:syms!45.15 191.10 178.50 128.04 341.30 // starting prices
n:2 // number of rows per update
flag:1 // generate 10% of updates for trade and 90% for quote
getmovement:{[s] rand[0.001]*prices[s]} // get a random price movement
// generate trade price
getprice:{[s] prices[s]+:rand[1 -1]*getmovement[s]; prices[s]}
getbid:{[s] prices[s]-getmovement[s]} // generate bid price
getask:{[s] prices[s]+getmovement[s]} // generate ask price
// timer function
.z.ts:{
 s:n?syms;
 $[0<flag mod 10;
 h(`upd;`quote;(n#.z.N;s;getbid'[s];getask'[s]));
 h(`upd;`trade;(n#.z.N;s;getprice'[s];n?1000))
 ];
 flag+:1;
 };
// trigger timer every 100ms
\t 100
```

This Feed Handler code is courtesy of the Kx Technical Whitepaper on:
https://code.kx.com/q/wp/building_real_time_tick_subscribers.pdf

websockets.html

Due to the length of code required for this example, the JavaScript and HTML code have been split into separate files

```
<!doctype html>
<html>
<head>
   <title>WebSocket PubSub Example</title>
</head>
<body onload="connect();">
   <!-- Create a section to filter on syms -->
   <section class="select">
      <h3>Select Syms: </h3>
      <div id="selectSyms"></div>
      <button type="submit" onclick="filterSyms();">Filter</button>
   </section>
   <!-- Set up placeholders to display the trade and quote outputs -->
   <section id="quotes" class="display">
      <div class="split">
         <h3>Quotes</h3>
         <table id="tblQuote"></table>
      </div>
      <div class="split">
         <h3>Trades</h3>
         <table id="tblTrade"></table>
      </div>
   </section>
   <!-- Load JavaScript files -->
   <script src="http://kx.com/q/c/c.js"></script>
   <script src="websockets.js"></script>

   <style>  /* define some CSS styling on page elements */
      section  {margin:10px;padding:20px;width:95%;}
      button {margin:10px;}
      h3 {margin:5px;}
      table {border-collapse:collapse;text-align:center;width:100%;}
      td,th {border:1px solid black;padding:5px 20px;width:25%}
      .split {float:left;width:45%;margin-right:20px;display:table;}
      #selectSyms {padding:10px;min-height:30px;}
   </style>
</body>
</html>
```

websockets.js

This script will be loaded into the web page by the HTML. Make sure this is saved as a .js file in the same directory as the above HTML file.

```
// initialise variable
var ws,syms = document.getElementById("selectSyms"),
     quotes = document.getElementById("tblQuote"),
     trades = document.getElementById("tblTrade");
function connect(){
   if ("WebSocket" in window) {
      ws = new WebSocket("ws://localhost:5001");
      ws.binaryType = 'arraybuffer'; // using serialisation
      ws.onopen=function(e){
   // on successful connection, we want to create an initial subscription
to load all the data into the page
         ws.send(serialize(['loadPage',[]]));
      };
      ws.onclose=function(e){console.log("disconnected");};
      ws.onmessage=function(e){
   // deserialise incoming messages
         var d =  deserialize(e.data);
   // messages should have format [‘function’,params]
   // call the function name with the parameters
         window[d.shift()](d[0]);
      };
      ws.onerror=function(e){console.log(e.data);};
   } else alert("WebSockets not supported on your browser.");
}
function filterSyms() {
  // get the values of checkboxes that are ticked and convert into an
array of strings
   var t = [], s = syms.children;
   for (var i = 0; i < s.length ; i++) {
      if (s[i].checked) { t.push(s[i].value); };
   };
  // call the filterSyms function over the WebSocket
   ws.send(serialize(['filterSyms',t]));
}
function getSyms(data) {
  // parse an array of strings into checkboxes
   syms.innerHTML = '';
   for (var i = 0 ; i<data.length ; i++) {
      syms.innerHTML += '<input type="checkbox" name="sym" value="' +
data[i] + '">' + data[i] + '</input>';
   };
}
function getQuotes(data) { quotes.innerHTML = tableBuilder(data); }
function getTrades(data) { trades.innerHTML = tableBuilder(data); }
function tableBuilder(data) {
  // parse array of objects into HTML table
   var t = '<tr>'
   for (var x in data[0]) {
      t += '<th>' + x + '</th>';
   }
```

```
   t += '</tr>';
   for (var i = 0; i < data.length; i++) {
      t += '<tr>';
      for (var x in data[0]) {
         t += '<td>' + (("time" === x) ?
data[i][x].toLocaleTimeString().slice(0,-3) : ("number" == typeof
data[i][x]) ? data[i][x].toFixed(2) : data[i][x]) + '</td>';
      }
      t += '</tr>';
   }
   return t ;
}
```

it's about time

Technical Whitepaper

5 C API for kdb+

Author:

Jeremy Lucid, who joined First Derivatives in 2013, is a kdb+ consultant based in London. He works for a global investment bank, specializing in real-time best execution analytics

CONTENTS

5.1 Introduction

In its traditional financial domain, and across an increasingly broad range of industries, one of the main strengths of kdb+ is its flexibility in integrating and communicating with external systems. This adoption enhancing feature is facilitated through a number of interfaces, including C and Java APIs, ODBC support, HTTP and WebSockets. This paper will illustrate how the C API can be used to enable a C program to interact with a kdb+ process, and so leverage the real-time streaming and processing strengths of kdb+. Particular consideration will be given to how the API can facilitate subscription and publication to a kdb+ tickerplant process, a core component of any kdb+ tick capture system. Just as trade and quote data can be piped into a financial kdb+ application through a C feedhandler, interesting new datasets in non- financial industries can readily be consumed and processed with minimal setup work in C.
For example, in a recent whitepaper exploring the application of kdb+ in the astronomy domain, http://code.kx.com/q/wp/kdb_in_astronomy.pdf, a standard scientific file format is loaded into kdb+ for fast calculation of the recessional velocities of celestial bodies.

While https://code.kx.com/q/interfaces/c-client-for-q/ should still be referred to as the primary source for up to date information on the C API, the examples presented in this paper are designed to form a complementary set of practical templates. These templates can be combined and leveraged to facilitate the application of kdb+ across a broad range of problem domains, and are made available for reference and reuse at https://github.com/kxcontrib/capi.

5.2 K Object

To begin understanding how the C API works, it is best to start by examining the contents of the k.h header file. This header file needs to be included in the C or C++ code in order to interact with a q process. In particular, the header includes the definition of the data type K, which is a pointer to the `k0` struct, shown below. This data type is important because all q types, such as atoms, lists, dictionaries and tables are encapsulated at the C level as instances of the K data type. Having K defined as a pointer is advantageous from a memory perspective, resulting in greater efficiency when passed as an argument to a C function.

```
/* Structure below corresponds to version 3.0 and higher */
typedef struct k0 {
    signed char m,a,t; // m,a are for internal use.
    C u;
    I r;
    union {
       G g;
       H h;
       I i;
       J j;
       E e;
       F f;
       S s;
       struct k0*k;
       struct {J n;G G0[1];};
    };
 } *K;
```

5.3 Connecting to a kdb+ process

5.3.1 Starting a q server

In order for a C program to interact with a q process using the API, it is necessary that the q process be enabled to listen for incoming connections. To start a q process in server mode, whereby it listens for requests on a particular port number, the process should be started from the terminal using the -p <port number> command line argument, as shown below.

```
$q -p 12345
q)\p    // Command used to show what port number the process is listening on
12345i
```

The process can now receive incoming connections via TCP/IP on port 12345. Alternatively, the port number can be (re)set using the command \p <port number> while within the q session.

```
$q q
)\p
0i       // Default value when no command line argument is given
q)\p 12345
q )\ p
12345i
```

Without setting a port number via the command line argument, the q session will default to a port value of 0i, meaning no socket listening. Explicitly setting the port to 0i, as shown below, will close any previously open port. This will result in all new connection attempts being rejected.

```
$q -p 12345
q)\p 0     // Close previously opened port
```

5.3.2 Opening a socket connection

In the k.h header file is defined the khpu function prototype, shown below, which can be used to open a network socket connection to a listening q process.

```
typedef int I;              // Already defined within k.h
typedef char *S;            // Already defined within k.h
I khpu(S hostname , I portNumber , S usernamePassword)
```

The function takes three parameters and returns the integer value associated with the connection handle. This integer value returned is the OS file descriptor. The first and second parameters expect the hostname and port number of the listening q process, respectively. The third parameter, usernamePassword, expects a string argument containing the username and password credentials required for authentication. The username and password in this string argument should be separated by a single colon. If the listening q process is not using authentication for incoming connections then the username and password values will be ignored.

```
khpu("localhost", 12345, "username:password");
```

The C program below shows how khpu can be used to create a socket handle to a q process already listening on port 12345. After the handle value is successfully error checked, using the convenience function handleOk (code given below), its value is printed to stdout and the handle closed using the kclose function.

```
/* File name: portopen.c */
#include "common.h"

int main()
{
  I handle;
  I portnumber= 12345;
  S hostname= "localhost";
  S usernamePassword= "kdb:pass";

  handle= khpu(hostname, portnumber, usernamePassword);
  if(!handleOk(handle))
    return EXIT_FAILURE;

  printf("Handle value is %d\n", handle);

  kclose(handle);
  return EXIT_SUCCESS;
}
```

Compile and Run

The c.o header file, required for compilation, can be found here
https://github.com/KxSystems/kdb/tree/master/l32

```
$gcc -DKXVER=3 -o portopen portopen.c c.o -lpthread
$ ./portopen
Handle value is 3
```

When khpu connects successfully the handle value returned is always a positive number. Positive values indicate success with negative values being reserved for error conditions. Possible return values for the khpu function are listed below, and form the basis for the error handling logic performed by handleOk. This function is one of a pair of error handing utilities which will be used across multiple subsequent examples. All utility functions are defined in the common.h header file which is loaded as part of each example. Particular attention should be paid towards error handling when using the API, and it is for this reason that such logic is included within examples presented herein. Further details on error signaling will be presented in section 3.4.

```
handle  >   0   "Active handle"
handle  =   0   "Authentication error"
handle  =  -1   "Error"
handle  =  -2   "Timeout" (When khpun is used instead of khpu)
```

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include "k.h"

I handleOk(I handle)
{
  if(handle > 0)
    return 1;
  if(handle == 0)
    fprintf(stderr, "Authentication error %d\n",
  handle); else if(handle == -1)
    fprintf(stderr, "Connection error %d\n",
  handle); else if(handle == -2)
    fprintf(stderr, "Time out error %d\n", handle);
  return 0;
}
```

The ”Timeout” error is associated with the khpun function which is similar to khpu but allows for a connection timeout to also be specified. The error -2 indicates that the specified timeout time was exceeded before the connection could be created. The function prototype for khpun is given below.

```
typedef  int  I ;      //  Already  defined  within k.h
typedef  char *S;      //  Already  defined  within k.h

I khpun(S  hostname,I  portNumber,S  usernamePassword,I  timeout)

//  Timeout  is  specified  in  units  of  milliseconds
//  A  5  second  timeout  would  require  a  timeout  value  of  5000
```

5.3.3 Running queries using the k function

The k function is used to send a message over a socket connection to a q process. Valid message instructions can be used to perform multiple actions, which include, but are not limited to, requests for data, table inserts, variable initialization and function execution. The response is encapsulated on the C
level as a K object and returned by the k function. Within the k.h header file the function is defined as follows.

```
typedef int  I ;
typedef char *S;

K  k(I  handle , const  S query ,...)
```

The first parameter to the k function is the handle value returned by the khpu function call. The second is the message string containing the instructions to be executed by the receiving q process. By definition, the ellipsis indicates the function can take an indeterminate number of arguments following the message string. These additional arguments would be the required inputs associated with execute instructions. For example, if the instruction is to execute the kdb+ `insert` function, the additional arguments required would be the table name and data object to be inserted, see 5.2 for such an
example.

Note that the k function requires the user to always input a sentinel value, `(K)0`, following the additional argument list. The presence of this sentinel is used as a control variable enabling the function to detect the end of the additional argument list. This value should also be present in the case where no additional arguments are required, see below.

```
// When no additional parameters are required the (K)0
// sentinel should be added as the third parameter
K k(handle , "query", K(0))

// When additional parameters are required , (K)0
// should be added following the last parameter
K k(handle , "query", param1 ,...,paramN , (K)0)
```

The k function returns an instance of a K object, from which an accessor function/macro can be used to access the data. Below is a simple example demonstrating how the k function can be used to confirm that a q process is 'alive'. Where 'alive' means that the socket connection can be established and that the server can respond to a trivial request. Such a test could be performed periodically to determine the q server state.

```
/* File name: alivecheck.c */
#include "common.h"

int main() {
  I handle;
  I portnumber= 12345;
  S hostname= "localhost";
  S usernamePassword= "kdb:pass";
  K result;

  handle= khpu(hostname, portnumber, usernamePassword);
  if(!handleOk(handle))
    return EXIT_FAILURE;
  printf("Handle value is %d\n", handle);
  result= k(handle, "2.0+3.0", (K) 0);
  if(isRemoteErr(result)) {
    kclose(handle);
    return EXIT_FAILURE;
  }
  printf("Value returned is %f\n", result->f);

  r0(result);
  kclose(handle);
  return EXIT_SUCCESS;
}
```

Compile and Run

```
$gcc -DKXVER=3 -o alivecheck alivecheck.c c.o -lpthread
$./alivecheck
Handle value is 3
Value returned is 5.000000
```

In the above example, another convenience function, `isRemoteErr`, is introduced to handle any errors encountered during the remote evaluation. The function is defined as follows and is explained in further detail in the following section.

```
/* common.h */

I isRemoteErr(K
  x) { if(!x) {
    fprintf(stderr, "Network error: %s\n",
    strerror(errno)); return 1;
  } else if(-128 == xt) {
    fprintf(stderr, "Error message returned : %s\n", x-
    >s); r0(x);
    return 1;
  }
  return 0;
}
```

5.3.4 Error Signaling

In general, there are two common types of errors; those generated during the socket handle initialization process and those which can occur during the evaluation of a request. As described in 3.2, where the error handling function `handleOk` was introduced, initialization errors can be identified by examining the integer value returned by the `khpu` and `khpun` function calls.

As an example of an initialization error, the ”Authentication Error” will occur when the user name or password credentials passed to `khpu` are invalid. In kdb+, a `q` process can be started in a restricted access mode where only users with valid credentials can connect, see `http://code.kx.com/q/wp/kdb_and_websockets.pdf`. Similarly, the ”Connection error”can result for incorrect hostname or port numbers being passed to `khpu`.

Evaluation errors occur from invalid queries being passed to the `q` process using `k`. In this case, the type element `(x->t)` of the `K` object, `x`, returned by `k` should be checked, see section 4.2. If the type value is 0 (null), then a network error has occurred. If the type value is -128, then `x->s` will give the error message resulting from the invalid query. In error.c, below, three invalid queries are passed to a q process and their associated error messages printed. For all subsequent examples in this paper, the function `isRemoteErr` will be used to capture such errors. For the complete set of error codes see `http://code.kx.com/q/ref/errors/#errors`.

```
/* File name: error.c */
#include "common.h"

int
  main()
  { K
  result
  ;
  I port= 12345;
  I timeout=
  5000; I
  handle;

  handle= khpun("localhost", port, "kdb:pass", timeout);
  if(!handleOk(handle))
    return EXIT_FAILURE;

  result= k(handle, "1+`2", (K) 0);
  // Handle network
  error if(!result) {
    perror("Network
    Error\n");
    kclose(handle);
    return EXIT_FAILURE;
  }

  if(-128 == result->t) {
    fprintf(stderr, "Error message returned : %s\n", result->s);
  }
  r0(result);

  result= k(handle, "`a`b`c=`a`b", (K) 0);
  // Handle network
  error if(!result) {
    perror("Network
    Error\n");
    kclose(handle);
    return EXIT_FAILURE;
  }
  if(-128 == result->t) {
    fprintf(stderr, "Error message returned : %s\n", result->s);
  }
  r0(result);

  result= k(handle, "select sym t :([] sym:`a`b)", (K) 0);
  // Handle network
  error if(!result) {
    perror("Network
    Error\n");
    kclose(handle);
    return EXIT_FAILURE;
  }
  if(-128 == result->t) {
    fprintf(stderr, "Error message returned : %s\n", result->s);
  }
  r0(result);

  kclose(handle);
  return
  EXIT_SUCCESS;
}
```

```
Compile and Run

$gcc  -DKXVER=3 -o  error  error.c  c.o -lpthread
$./error
Error  message  returned:  type
Error  message  returned:  length
Error  message  returned:  from
```

5.3.5 Synchronous vs Asynchronous communication

If a C process is using the API to publish messages to a q process, such as a tickerplant, then either a synchronous or asynchronous call can be used. The sign of the first parameter to the k function determines if the function call will execute synchronously or asynchronously. In the example above, k executed synchronously because the handle value passed to the function was a positive value. The synchronous call will block until a response is received, which is appropriate if the sending process requires an acknowledgement, however, it will reduce the potential message rate. In order to send an asynchronous message, the k function should be passed the negative handle value.

```
k(-handle ,  "1.0  +  2.0",  (K)0);
```

The asynchronous option is recommended when maximum data throughput is desired and the sender does not require an acknowledgment. Greater technical details on synchronous vs asynchronous requests can be found in the following whitepapers,
http://code.kx.com/q/wp/common_design_principles_for_kdb_gateways.pdf

http://code.kx.com/q/wp/query_routing_a_kdb_framework_for_a_scalable_load_balanced_system.pdf

5.4 Extracting data from a K object

5.4.1 Accessing a K object element

As demonstrated in the previous chapter, the k function can be used to send messages over a socket connection to a q process. The data returned from the q process is then encapsulated at the C level as a K object. That is to say, any q object such as an atom, list, dictionary or table will be contained within the structure referenced by the K pointer. To extract data from a K object, the -> operator can be used to access the member elements of the k0 structure.

5.4.2 Object type element t

Given a K object x, the member element x->t identifies the objects data type. This element can also be accessed using the equivalent macro provided, xt.

```
#define  xt  x->t
```

The following reference table gives the q data type (or error) for the given t value.

```
//t value             Object type or error
t < 0                 Scalar (atom) or error type
t = 98                table
t = 99                Dictionary or keyed table
t = -128              Error
t > 0                 List where elements are of same data type
t within 0 19         List
t within 20 76        Enumerates types
t within 77 97        Function Types
t within 100 112      Function types
```

5.4.1 List length element n

The element n indicates the length of a list object. The list may contain atomic elements or other K objects. Given a K object x, the n member element can be accessed using the notation x->n. The example below demonstrates usage for the case of a list containing atomic float elements.

```
/* File name: length.c */
#include "common.h"

int main() {
  I handle;
  I portnumber= 12345;
  S hostname= "localhost";
  S usernamePassword= "kdb:pass";
  K list;

  handle= khpu(hostname, portnumber, usernamePassword);
  if(!handleOk(handle))
    return EXIT_FAILURE;

  list= k(handle, "2*1 2 3f", (K) 0);
  if(isRemoteErr(list)) {
    kclose(handle);
    return EXIT_FAILURE;
  }

  printf("List length is %lld\n", list->n);

  r0(list);
  kclose(handle);
  return EXIT_SUCCESS;
}
```

Compile and Run

```
$gcc  -DKXVER=3 -o   length   length.c c.o -lpthread
$./length
List  length  is 3
```

For more complex data structures such as dictionaries, n indicates the number of keys and the number of values associated with each key. Similarly, for tables, n can be used to extract the number of rows and columns present. Working examples for dictionaries and tables are given further below

5.4.2 Extracting atoms

From the reference table above, when the value of t is less than 0 and not equal to -128 then the object is an atom (or scalar). In which case, the value of t identifies the q data type, using the lookup table below. Corresponding to each data type is the associated accessor function used to extract
the scalar value from the K object. The accessor functions access the union member elements of the structure.

```
Note that the type numbers below are given for vectors (positive values).
By convention, for atoms (single elements) the type number is negative.
For example: -6 is the number type of a atom int value.

Data Type       Accessor         Type number          Type Name
mixed list      kK               0                    -
boolean         x->g             1                    KB
guid            x->U             2                    UU
byte            x->g             4                    KG
short           x->h             5                    KH
int             x->i             6                    KI
long            x->j             7                    KJ
real            x->e             8                    KE
float           x->f             9                    KF
char            x->i             10                   KC
symbol          x->s             11                   KS
timstamp        x->j             12                   KP
month           x->i             13                   KM
date            x->i             14                   KD
datetime        x->f             15                   KZ
timespan        x->j             16                   KN
minute          x->i             17                   KU
second          x->i             18                   KV
time            x->i             19                   KT
table/flip      x->k             98                   XT
dict/table with kK(x)[0] (keys)  99                   XD
primary key     kK(x)[1] (values)
```

5.4.3 Extracting simple lists

When the value of t is within the range 0 to 19 the object is a simple list. A simple list is analogous to a C array in which each element in the structure has the same data type. As is the case with atoms, the value of t identifies the data type of the simple list elements. To access the list values, the following accessors are provided.

```
#define kG(x)  ((x)->G0)
#define kC(x)  kG(x)
#define kH(x)  ((H*)kG(x))
#define kI(x)  ((I*)kG(x))
#define kJ(x)  ((J*)kG(x))
#define kE(x)  ((E*)kG(x))
#define kF(x)  ((F*)kG(x))
#define kS(x)  ((S*)kG(x))
#define kK(x)  ((K*)kG(x))
```

The following example demonstrates use of these functions for the case of a simple list containing ints, floats and symbols.

```
/* File name: lists.c */
#include "common.h"

int
  main()
  { I
  handle
  ;
  I portnumber= 12345;
  S hostname= "localhost";
  S usernamePassword=
  "kdb:pass"; K x;

  handle= khpu(hostname, portnumber, usernamePassword);
  if(!handleOk(handle))
    return EXIT_FAILURE;

  x= k(handle, "reverse `A`B`C", (K) 0);
  if(isRemoteErr(x)) {
    kclose(handle);
    return
    EXIT_FAILURE;
  }

  printf("Symbols: %s %s %s\n", kS(x)[0], kS(x)[1], kS(x)[2]);
  r0(x);
  x= k(handle, "reverse 1 2 3", (K) 0);
  if(isRemoteErr(x)) {
    kclose(handle);
    return
    EXIT_FAILURE;
  }

  printf("Ints: %lld %lld %lld\n", kJ(x)[0], kJ(x)[1],
  kJ(x)[2]); r0(x);
  x= k(handle, "reverse 1.0 2.0 3.0", (K) 0);
  if(isRemoteErr(x)) {
    kclose(handle);
    return
    EXIT_FAILURE;
  }

  printf("Floats: %f %f %f\n", kF(x)[0], kF(x)[1], kF(x)[2]);
  r0(x);

  kclose(handle);
  return
  EXIT_SUCCESS;
}
```

Compile and Run

```
$gcc -DKXVER=3 -o lists lists.c c.o -lpthread */
$./lists
Symbols:   C B A
Ints:  3  2  1
Floats:   3.000000  2.000000  1.000000
```

5.4.4 Extracting mixed lists

In the case where `x->t` is exactly zero, the K object contains a mixed list of other K objects. Each element in the list is a pointer to another K object. In the example below, a mixed list containing two elements is returned from the q process. Element 1 is a list of 5 integers and element 2 is a list of 3 floats. The example demonstrates how the length and type of each list element can be determined along with how the values can be extracted.

```
/* File name: mixedList.c */
#include "common.h"

int
  main()
  { J i;
  I handle;
  I portnumber= 12345;
  S hostname= "localhost";
  S usernamePassword= "kdb:pass";
  K mixedList, elementOne, elementTwo;

  handle= khpu(hostname, portnumber, usernamePassword);
  if(!handleOk(handle))
    return EXIT_FAILURE;

  mixedList= k(handle, "(1 2 3 4 5;20.0 30.0 40.0)", (K) 0);
  if(isRemoteErr(mixedList)) {
    kclose(handle);
    return
    EXIT_FAILURE;
  }

  printf("Object mixed list type: %d\n", mixedList->t);
  printf("Mixed list contains %lld elements\n", mixedList->n);

  elementOne=
  kK(mixedList)[0];
  elementTwo=
  kK(mixedList)[1];

  printf("elementOne contains %lld elements\n", elementOne-
  >n); printf("elementTwo contains %lld elements\n",
  elementTwo->n);

  for(i= 0; i < elementOne->n; i++) {
   printf("elementOne[%lld] = %lld\n", i, kJ(elementOne)[i]);
  }

  for(i= 0; i < elementTwo->n; i++) {
    printf("elementTwo[%lld] = %f\n", i,
    kF(elementTwo)[i]);
  }

  r0(mixedList);
  kclose(handle);
  return
  EXIT_SUCCESS;
}
```

Compile and Run

```
$gcc -DKXVER=3 -o mixedList mixedList.c c.o -lpthread
$./mixedList
Object mixed list type : 0
Mixed list contains 2 elements
elementOne contains 5 elements
elementTwo contains 3 elements
elementOne has type 7
elementTwo has type 9
elementOne [0]=1
elementOne [1]=2
elementOne [2]=3
elementOne [3]=4
elementOne [4]=5
elementTwo [0]=20.000000
elementTwo [1]=30.000000
elementTwo [2]=40.000000
```

5.4.5 GUID

Since kdb+ version 3.0, the globally unique identifier (GUID) data type is supported. This is a 16 byte data type which can be used for storing arbitrary 16 byte values, typically transaction ids. In general, since v3.0, there should be no need to use char vectors for ids. Instead id's should be int, sym or GUID. For table lookups, GUIDs are also much faster when performing equality comparisons in the where clause relative to the string representation, which also takes up 2.5 times more space. The memory saving can be easily illustrated by comparing the byte size of a table containing a single column of GUIDs with that of a table containing a single column string equivalent. The internal q function, `-22!`, conveniently returns the table byte size. [Compute Ratio: `q)%[;] . -22!'(([]string 100?0Ng);([]100?0Ng))`].
The example below demonstrates how to extract a single or list of GUIDs from a q process.

```
/* File name: guid.c
*/#include "common.h"

int main() {
  J i, j;
  I handle;
  I portnumber= 12345;
  S hostname= "localhost";
  S usernamePassword= "kdb:pass";
  K singleGuid, multiGuid;

  handle= khpu(hostname, portnumber, usernamePassword);
  if(!handleOk(handle))
    return EXIT_FAILURE;

  singleGuid= k(handle, "rand 0Ng", (K) 0);
  if(isRemoteErr(singleGuid)) {
    kclose(handle);
    return EXIT_FAILURE;
  }

  printf("Single guid: type %d\n", singleGuid->t);
  printf("Single guid: length %lld\n", singleGuid->n);

  for(i= 0; i < 16; i++) {
    printf("%02x", kU(singleGuid)->g[i]);
  }
  r0(singleGuid);
  printf("\n");

  multiGuid= k(handle, "2?0Ng", (K) 0);
  if(isRemoteErr(multiGuid)) {
    kclose(handle);
    return EXIT_FAILURE;
  }

  printf("Multi guid: type %d\n", multiGuid->t);
  printf("Multi guid: length %lld\n", multiGuid->n);

  for(i= 0; i < multiGuid->n; i++) {
    for(j= 0; j < 16; j++) {
      printf("%02x", kU(multiGuid)[i].g[j]);
    }
    printf("\n");
  }
  r0(multiGuid);
  kclose(handle);
  return EXIT_SUCCESS;
}
```

```
Compile and Run

$gcc  -DKXVER=3 -o  guid  guid.c  c.o  -lpthread  */
$./guid
Single  guid:  type     -2 Single  guid:  length        1
be6cc222695dd8964e46c709b0
6f3573 MultiGuid:  type  2
MultiGuid:  length      2
730690ccf4ebd95c6d2d039255d
11ae6 45135fc5509403e4ab7eae7
05ba54437
```

5.4.6 Dictionaries

In kdb+, a dictionary is a data type which creates an association between a list of keys and a list of values. From the lookup table in section 4.2, its numerical type, `x->t`, has value 99 (encoded type name XD). In the next example, the `k` function is used to return a simple dictionary object, `x`. As expected, `x` contains a list of length two, where the first element corresponds to the dictionary keys, and the second corresponds to the dictionary values. In particular, the keys element has value 11 (encoded type name KS), indicating that the keys are a symbol list, see 4.4. The second element has type 0, indicating that the values of the dictionary are contained in a mixed list. Therefore, not all keys have an associated value of the same data type. Below, the mixed list (`values` is iterated through to extract and display the individual elements of each simple list element.

```c
/* File name: dict.c */
#include "common.h"

int main() {
  J i;
  I handle;
  I portnumber= 12345;
  S hostname= "localhost";
  S usernamePassword= "kdb:pass";
  K x, keys, values;

  handle= khpu(hostname, portnumber, usernamePassword);

  if(!handleOk(handle))
    return EXIT_FAILURE;

  x= k(handle, "`a`b`c!((1;2;3);(10.0;20.0);(`AB`CD`EF`GH))", (K) 0);

  if(isRemoteErr(x)) {
    kclose(handle);
    return EXIT_FAILURE;
  }

  printf("Dict type = %d\n", x->t);
  printf("Num elements = %lld (keys and values)\n", x->n);

  keys= kK(x)[0];
  values= kK(x)[1];

  printf("Keys type = %d\n", keys->t);
  printf("Num keys = %lld\n", keys->n);

  printf("Values type = %d\n", values->t);
  printf("Num values = %lld\n", values->n);
```

```
    for(i= 0; i < keys->n; i++) {
        printf("keys[%lld] = %s\n", i, kS(keys)[i]);
      }

    for(i= 0; i < values->n; i++) {
        printf("values[%lld] has length %lld\n", i, kK(values)[i]->n);
      }

      J               *values0=
      kJ(kK(values)[0]);       F
      *values1=
      kF(kK(values)[1]);       S
      *values2=
      kS(kK(values)[2]);

      printf("values[0;] = %lld %lld %lld\n", values0[0], values0[1],
      values0[2]); printf("values[1;] = %lf %lf     \n", values1[0],
      values1[1]); printf("values[2;] = %s %s %s %s  \n", values2[0],
      values2[1], values2[2],
             values2[3]);

      r0(x);
      kclose(handle);
      return
      EXIT_SUCCESS;
    }
```

Compile and Run

```
$gcc  -DKXVER=3 -o  dict  dict.c  c.o  -lpthread  */
$./dict
Dict  type =  99
Num  elements  =  2  (keys  and  values)
keys  type =  11
Num  keys   =  3
Values type =     0
Num Values   =    3
keys [0]  =  a
keys [1]  =  b
keys [2]  =  c
values [0]  has  length  3
values [1]  has  length  2
values [2]  has  length  4
values [0;]  - 1 2  3
values [1;]  -  10.000000  20.000000
values [2;]  -  ABC  DEF  GHI  JKL
```

5.4.7 Tables

Extracting data from a K object containing a table is similar to that for a dictionary. The element `x->k` will contain a two element list, representing a dictionary of columns to values. The first element, `(kK(x->k)[0])`, corresponds to the column names, with the second, `(kK(x->k)1])`, corresponding to the values. Table keys are always symbol vectors (KS). The number of columns present in the table is easily determined using the n element (see `columns->n` below). The values object is a mixed list, where each element in the list is another K object containing a simple or mixed list. The example below shows how the column names and values of a kdb+ table can be extracted.

```c
/* File name: table.c */
#include "common.h"

int
  main()
  { J i;
  I handle;
  I portnumber= 12345;
  S hostname= "localhost";
  S usernamePassword= "kdb:pass";
  K table, columns, values, col1, col2;

  handle= khpu(hostname, portnumber, usernamePassword);
  if(!handleOk(handle))
    return EXIT_FAILURE;

  // Execute a query which performs an aggregate on table t
  // Table t can be defined on the q process as follows
  // q)t:([] sym:`a`a`b`b`c`c;price:1.0 2.0 3.0 4.0 5.0 6.0)
  table= k(handle, "select from t where price=(max;price) fby sym", (K) 0);
  if(isRemoteErr(table)) {
    kclose(handle);
    return
    EXIT_FAILURE;
  }

  // Extract columns and values
  elements columns= kK(table->k)[0];
  values= kK(table->k)[1];

  // Show column count and print column names
  printf("Number of Columns = %lld\n", columns->n);
  printf("Columns: %s %s\n", kS(columns)[0], kS(columns)[1]);

  // Access the table values
  printf("values object has type = %d\n", values->t);

  col1=
  kK(values)[0];
  col2=
  kK(values)[1];

  printf("Number of elements in column 1 is %lld\n", col1->n);

  for(i= 0; i < col1->n; i++) {
   printf("%s %lf\n", kS(col1)[i], kF(col2)[i]);
  }

  r0(table);
  kclose(handle);
  return
  EXIT_SUCCESS;
}
```

Compile and Run

```
$q
q)t:([]  sym:`a`a`b`b`c`c;price:1.0  2.0  3.0  4.0  5.0  6.0)
q)\p  12345
$gcc  -DKXVER=3 -o  table  table.c  c.o  -lpthread
$./table
Number of Columns  = 2
Columns: sym price
values  object  has  type  =  0
This means it  contains a  mixed  list  of  other  K objec ts
Number  of  elements  in  row1  is  3
a      2.000000
b      4.000000
c      6.000000
```

5.5 Creating K objects

In the code example alivecheck.c, the query evaluated on the q process was very simple, "2.0 + 3.0", and didn't require additional arguments. However, if the query were of a functional form, requiring a single scalar parameter argument, for example `func:{[x]1.0 + x}` , the scalar argument should be input immediately after the query. For the argument to be passed successfully, it is necessary that the C variable be converted to its corresponding K object. The examples below show how single or multiple arguments of different data types can be passed to the functional query using such

```
// Pass a single integer argument
result = k(handle ,"{[x] 1 + x}", ki(2), (K)0);

// Pass a single float argument
result = k(handle ,"{[x] 1.0 + x}", kf(2.0), (K)0);

// Pass multiple float arguments
result = k(handle ,"{[x;y] x + y}", kf(1.0), kf(2.0), (K)0);

// Pass a single symbol argument
result = k(handle ,"{x in `ABC `DEF `GHI}", ks((S)"ABC"), (K)0);
```

conversion functions.

The table below shows the list of functions used to perform this conversion for other data types not given in the examples above.

```
K ka(I);          //Create an atom of type
K kb(I);          //Create a boolean
K ku(U);          //Create a guid
K kg(I);          //Create a byte
K kh(I);          //Create a short
K ki(I);          //Create an int
K kj(J);          //Create a long
K ke(F);          //Create a real
K kf(F);          //Create a float
K kc(I);          //Create a char
K ks(S);          //Create a symbol
K ktj(-KP,J);     //Create a timestamp
K kt(I);          //Create a time
K kd(I);          //Create a date
K ktj(-KN,J);     //Create a timespan
K kz(F);          //Create a datetime
```

5.5.1 Creating and passing vector arguments

It is common that a function call or query require a vector as opposed to a scalar argument. An example of a vector argument in kdb+ would be a simple or mixed list. To create a list object, be it simple or mixed, using the API library, the following functions are available in the k.h header file.

```
//Create a simple list, where type does not take value zero
K ktn(I type,I length);

// Create a mixed list
K ktn(0,I length)
K knk(I length,...); // Common and useful shortcut
```

Both of these functions will be described in greater detail further below.

5.6 Creating a simple list object

A simple list in kdb+ is a collection of elements each with the same data type, analogous to a C array. To create a list object, using the API, the function `ktn` is used. The first parameter to this function specifies the data type of the list elements. From the reference table shown at the top of the code.kx reference page, http://code.kx.com/q/interfaces/c-client-for-q/, either the kdb+ type number or encoded type name values can be used to specify the data type. The second parameter specifies the length of the list. The example below shows how to create an integer list object of length 2.

```
// Encoded type name is used to specify the type.
// List length initialised to 2
K integerList = ktn(KI, 2);
```

With the list object initialized it is straightforward to insert elements into the list using the Interface list accessor function, as given in the reference table. In the example below, it is demonstrated how the elements of the list can be initialized using the `kI` accessor function. This two element list is then passed as a parameter to an insert function on a q process. The list is inserted as a new row into the table `t`.

```
I i = 1;
I j = 2;
// List of type int initialised with two elements allocated

K integerList = ktn(KI,2);

kI(integerList)[0]=i;// Insert element i to position 0

kI(integerList)[1]=j;// Insert element j to position 1

k(handle, "insert",ks((S) "t"), integerList, (K)0);
```

If it is not possible to know in advance the length of the list object, an alternative approach would be to initialize a list object with length 0 and subsequently append elements to the list using the provided join functions shown below.

```
typedef void V;
typedef char *S;

K ja(K*,V*);     // Join an atom to a list
K js(K*,S);      // Join a string to a list
K jk(K*,K);      // Join another K object to a list
K jv(K*,K);      // Join another K list to the first
```

The example below shows how an integer list of size zero can be initialized and later appended to using the `ja` function. Note that because the first and second parameters to this function are of type pointer (*), the value passed needs to be the memory address of the given object.

```
I i
= 1;
I j
= 2;

//  List  of  type  KI (int) initialized with size
zero K integerList =  ktn(KI, 0);

integerList = ja (& integerList ,& i ); // Append element i to the

list integerList = ja(&integerList ,&j);  // Append element j to the

list k(handle, "insert",ks((S) "t"), integerList, (K)0);
```

5.6.1 Creating a mixed list object

A mixed list in kdb+ is a collection of elements where each element may be of a different data type, analogous to a C struct. To create a mixed list object, using the API, the function knk is used. From the definition above, the knk function is defined as a variable argument function meaning the number of parameters is not fixed, as shown below.

```
//  Create  a  mixed  list ,  where  n  is  the  number  of  elements
typedef  int  I;
K  knk(I  n ,...);
```

Above n defines the number of elements in the list, where each list element, 1 to n, may be of a different data type. In the example below, a mixed list object of length 3 is created and initialized with

elements of a different data type; symbol, float and integer. This example demonstrates how a mixed list can be passed to the k function for insertion into a table. In this case, the mixed list is inserted as a single row into the table trade.

```
K  mixedList;
mixedList  =  knk(3,  ks((S)"ABC"),  kf(10.0) ,  ki (20));

k(handle,"{[x] `trade  insert  x}",  mixedList ,  (K)0);
```

A mixed list can also be created where each element is itself a simple list. This object is typically used for bulk inserts in which multiple rows are inserted at once.

```
multipleRow  =  knk(3, ktn(KS,n), ktn(KF,n), ktn(KI,n))
```

The below code segment demonstrates how a mixed list object can be populated with simple lists, each of length two.

```
I int_1  =  1;
I int_2 =  2;
K  integerList  =  ktn(KI,  2);  //  Simple  list  object  of  type  int
kI(integerList)[0]  = int_1;
kI(integerList)[1]  =  int_2;

F  float_1  = 1.5;
F  float_2  =  2.5;
K  floatList  =  ktn(KF,  2);   //  Simple  list  object  of  type  float
kF(floatList)[0]  = float_1;
kF(floatList)[1]  =  float_2;

S  sym_1=ss("IBM");
S  sym_2=ss("ABC");

K  symList  =  ktn(KS,  2);      //  Simple  list  object  of  type  symbol
kS(symList)[0]  =  sym_1;
kS(symList)[1] = sym_2;
//  2  rows  with 3 columns
multipleRow  = knk(3, symList , floatList , integerList);
```

5.7 Memory Management

Whereas kdb+ manages a program's memory requirements automatically, C provides several functions for manual memory allocation and management. These functions include `malloc`, `calloc`, `free`, and allow for fine tuned memory management: an important requirement for software destined for mobile or miniature devices. In the case where a C program is required to handle K objects, the C API provides a few simple functions, detailed below, to manage memory through the process of incrementing or decrementing a
K objects reference count.

5.7.1 Reference counting

kdb+ automatically manages reference counts to determine when an object is safe to destroy. As the number of references to an object increase or decrease, its associated reference count is incremented or decremented accordingly. When the reference value reaches zero the object can be safely destroyed. In the case of a memory resource, when the object is destroyed the memory would be freed and made available for reallocation. In kdb+, the reference count can be accessed directly using the internal function `-16!`. The following example shows the reference count incrementing and decrementing as expected.

```
q)a:5?10.0
q)a
7.85033  5.347096  7.111716  4.11597  4.931835
q)-16!a
1i
q)b: a
q)-16!a
2i
q)-16!b
2i
q) delete  a  from  `.
q)-16!b   //  Since  a  was  deleted , b 's  ref  count  was  decreased
1i
```

To increment or decrement the reference count manually within a C program the following API functions are available.

```
/* Increment the object's reference count */ r1(K x)
/* Decrement the object's reference count */ r0(K x)
```

The member element `r` stores the current reference count for a K object.

```
/* Fine name: refcount.c */
#include "common.h"

int main() {
  K a, b;
  khp("", -1); // initialise memory if not opening connection before allocation
  a= ki(3);    // Create K object a
  printf("New K object a has ref count = %d\n", a->r);

  b= a; // Set b equal to a
  r1(a); // Increment a's reference count
  printf("incremented ref count of a = %d\n", a->r);
  printf("ref count of b = %d\n", b->r);

  r0(b); // Decrement b's reference count
  printf("decrement ref count of b = %d\n", b->r);
  printf("ref count of a was decremented to = %d\n", a->r);

  printf("value of a is still available: %d\n", a->i);

  r0(a); // When an object of ref count 0 passed to r0, the object is destroyed
  return EXIT_SUCCESS;
}
```

Compile and Run

```
$gcc -DKXVER=3 -o refcount refcount.c c.o -lpthread
$./refcount
New K object a has ref count = 0
incr emen ted ref count of a = 1
ref count of b = 1
decr emen ted ref count of b = 0
ref count of a was decremented to 0
value of a is still available : 3
```

One needs to be very careful in the case where a K object is being passed as an argument to the k function, but is still required within the C program thereafter. This is because the objects reference count will get decremented during the k function return. If the objects reference count was zero prior to being passed, the object will be destroyed. This behavior is seen clearly in the example below, where the integer value held in the K object `a` is no longer available after `a` is passed as an argument to k.

```c
/* File name: refcountk.c */
#include "common.h"

int
  main()
  { I
  handle
  ;
  I portnumber= 12345;
  S hostname= "localhost";
  S usernamePassword=
  "kdb:pass"; K response, a;
  // NOTE: connection must be opened before creating K objects
  handle= khpu(hostname, portnumber, usernamePassword);
  if(!handleOk(handle))
    return EXIT_FAILURE;

  a= ki(5);
  printf("ref count of a is %d\n", a->r);
  printf("int value of a is %d\n", a->i);

  response= k(handle, "{x+3i}", a, (K) 0);
  // it is illegal to access a after this call as object has been destroyed


  if(isRemoteErr(response
    )) { kclose(handle);
   return EXIT_FAILURE;
  }
  printf("response value is %d\n", response->i);

  r0(response);
  kclose(handle);
  return
  EXIT_SUCCESS;
}
```

```
Compile and Run

$gcc -DKXVER=3 -o refcountk refcountk.c c.o -lpthread
$./refcountk
ref count of a is 0
int value of a is 5
response value is 8
```

To ensure a K object is not destroyed as a result of this behavior, simply increment the objects reference count, using `r1` as described, prior to calling `k`.

In addition to the before mentioned memory management functions, there are also a number of functions which are required for multi-threaded program execution, such as `m9` and `setm`. Detailed usage examples of these are outside the scope of this paper, however, information can be found on the code.kx reference page, see
http://code.kx.com/q/interfaces/c-client-for-q/#managing-memory-and-reference-counting

5.8 Shared C libraries for kdb+

In kdb+, it is possible to dynamically load shared libraries to extend functionality at run time. Being able to create libraries and taking advantage of pre-existing libraries is a good way to accelerate development and enable multiple applications to share common modules. To create user-defined functions in C which can be loaded and used by a q process, the function should be defined such that the input parameters and return value are K objects.

```
K myfunc (K param1 ,... , K paramN )
{
 K  result;

 return  result;
}
```

5.8.1 Loading a shared library using 2:

The `2:` function is used to dynamically import C functions from a shared library into a q process. It is a dyadic function where arguments can be passed using either the postfix or infix notation as shown below.

```
2:[ argumentOne;argumentTwo]

or

argumentOne 2:  argumentTwo
```

Argument one is a symbol representing the name of the shared dynamic library (excluding .so extension) which should in the same location as your q executable. Argument two is a list containing two elements. The first of these elements is a symbol corresponding to the function name, and the second is the integer number of arguments that function accepts. For a shared library, `myLib.so`, and single argument library function, `myfunc`, loading would look as follows.

```
q)f:`myLib 2: (`myfunc ;1)

//  If  the  shared library is not located in the same folder q
//  as  the  executable , it can be loaded by giving the
//  explicit  path

q)f:`:/path/to/myLib  2:  (`myfunc ;1)
```

5.8.2 User defined function - Dot product

Below is an example of a user defined function to compute the dot product of two vectors. The function first performs data type and length checks on the arguments passed before continuing to the calculation. If these checks fail, the `krr` function can be used to return the error back to the calling q process.

```
/* File name: mathLib.c */
#include <stdio.h>
#include "k.h"

K dotProduct(K x, K
  y) { J i;
  double result= 0.0;
  // Check data types are
  consistent if(x->t != KF || y-
  >t != KF) {
   return krr("type");
  }

  // Check vector lengths are
  equal if(x->n != y->n) {
    return krr("length");
  }

  for(i= 0; i < x->n; i++) {
    result+= kF(x)[i] *
    kF(y)[i];
  }

  return kf(result);
}
```

Below the `mathLib.c` file is compiled into a shared library and subsequently loaded into a q session. Two simple lists, a and b, of type float are created and the dot product computed using the loaded function. The result can be verified using the in-build kdb+ dot product function, $.

Compile and Run

```
$gcc -DKXVER= 3 -c  -Wall  -Werror  -fpic  mathLib.c  -lpthread
$gcc -DKXVER= 3 -shared  -o  mathLib.so  mathLib.o  -lpthread
$q
q)dotProduct:`mathLib 2:(`dotProduct;2)
q)a:5?10.0
q)b:5?10.0
q)a
3.927524 5.170911 5.159796 4.066642 1.780839
q)b
3.017723 7.85033 5.347096 7.111716 4.11597
q)a$b
116.2861
q)dotProduct[a;b]
116.2861
q)b:6?10.0
q)dotProduct[a;b]
'length
  [0]  dotProduct[a;b]
       ^
q)b:5?10
q)dotProduct[a;b]
'type
  [0]  dotProduct[a;b]
       ^
q)
```

5.8.3 Hashing algorithms - md5 and sha256

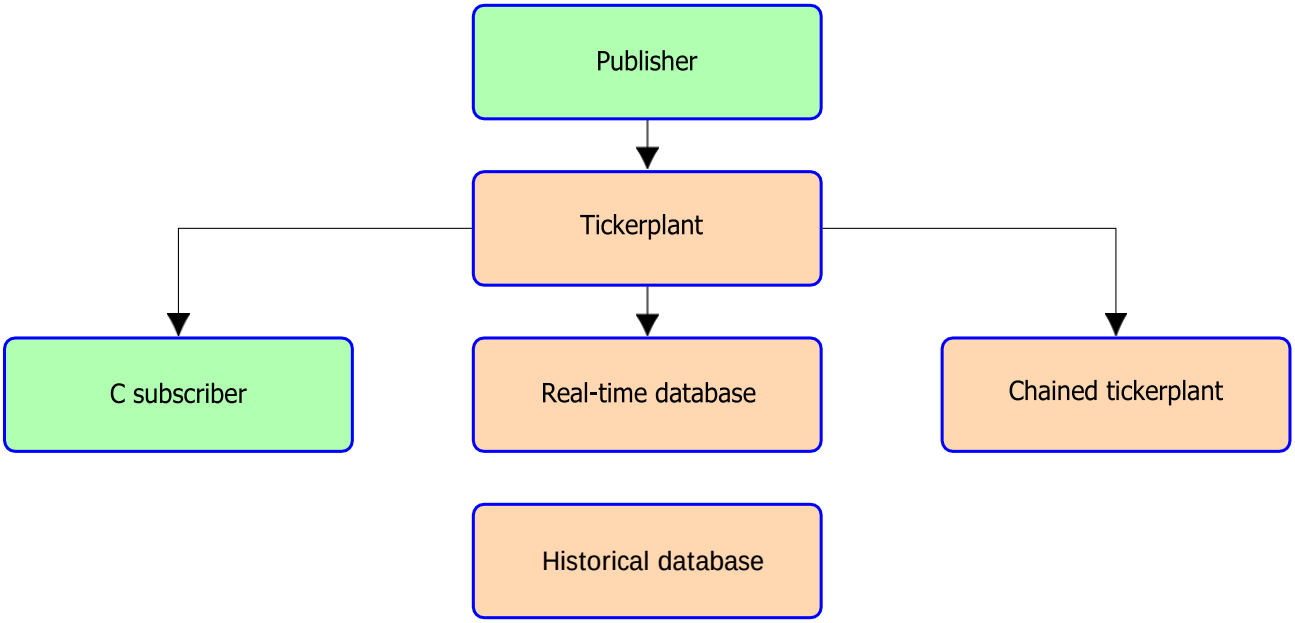
The following example demonstrates how to create a shared library, `cryptoLib.so`, which make use of the open-source openssl library. OpenSSL is an open source project which contains general purpose cryptographic tools, shown below. The first user-defined C function, `md5`, uses the library to compute the
md5 hash (https://en.wikipedia.org/wiki/MD5) of a given string input. This function is already implemented natively in kdb+ and so can be used to validate the functions operation. The second C function, `sha256`, uses the library to compute the more secure hashing algorithm, https://en.wikipedia.org/wiki/SHA-2, which is not native to kdb+.

```
/* File name: cryptoLib.c */
#include <stdlib.h>
#include <string.h>
#include <openssl/md5.h>
#include <openssl/sha.h>
#include "k.h"

K md5(K inputString) {
  K outputHash;
  if(KC != inputString->t)
    return krr("type");
  outputHash= ktn(KG, MD5_DIGEST_LENGTH);
  MD5(kC(inputString), inputString->n, kG(outputHash));
  return outputHash;
}

K sha256(K inputString) {
  K outputHash;
  if(KC != inputString->t)
    return krr("type");
  outputHash= ktn(KG, SHA256_DIGEST_LENGTH);
  SHA256(kC(inputString), inputString->n, kG(outputHash));
  return outputHash;
}

K sha512(K inputString) {
  K outputHash;
  if(KC != inputString->t)
    return krr("type");

  outputHash= ktn(KG, SHA512_DIGEST_LENGTH);
  SHA512(kC(inputString), inputString->n, kG(outputHash));
  return outputHash;
}
```

```
Compile and Run

$gcc -DKXVER=3 -c -fpic cryptoLib.c -lpthread -lcrypto -lssl
$gcc -DKXVER=3 -shared -o cryptoLib.so cryptoLib.o -lpthread -lcrypto -lssl
$q
q)cmd5:`cryptoLib 2:(`md5;1)
q)cmd5"kx tech"
0x78fb4fda39a5b41ec9ffb5fdbdd839
40 q)md5"kx tech"
0x78fb4fda39a5b41ec9ffb5fdbdd839
40 q)csha256:`cryptoLib
2:(`sha256;1) q)csha256"kx tech"
0xa92b19579d02ed9ebde54951d3....
```

5.9 Subscribing to a kdb+ tickerplant

A kdb+ tickerplant is a q process specifically designed to handle incoming, high-frequency, data feeds from publishing processes. The primary responsibility of the tickerplant is to manage subscription requests and publish data quickly to its subscribers. In the vanilla kdb+ setup, illustrated below, the
Real-time database (RDB) and chained tickerplant q processes are the most common type of subscriber, however, C applications are also possible using the API.



For further information regarding the above vanilla setup, please refer to the following whitepaper devoted to the subject,
http://code.kx.com/q/wp/building_real_time_tick_subscribers.pdf.

5.9.1 Test tickerplant and feedhandler setup

To facilitate testing of a C subscriber process, the following kdb+ tickerplant will be used, https://github.com/KxSystems/kdb-tick/blob/master/tick.q

For the sake of demonstration, the publisher will be configured to send a mock market data feed in the form of trade records to the tickerplant. The trade table schema to be used is defined below.

```
/* File name: trade.q */
/* File should be stored in the tick folder */
trade:([]
  time:`timespan$();
  sym:`symbol$();
  price:`float$();
  size:`long$()
 )
```

The tickerplant process is started from the command line as follows.

```
$q  tick.q  trade  C:/logs/tickerplant/  -p  5010
```

Above, the first argument following tick.q is the name of table schema file to use. The second argument is the location where the tickerplant log file will be created and the value following the -p flag is the port the tickerplant will listen on. The C process will use this port number when initializing the connection. The final step in this setup is to create a kdb+ mock feedhandler process which will act as the trade data source for the tickerplant. Below is a simple publishing process which is sufficient for the demonstration.

```
/* File name: feed.q */ /* Run: feed.q */

h:hopen 5010
syms:`VOD`IBM`APL

publishtrade:{[]
   nrow:first 1?1+til 4;
   h(".u.upd";`trade;(nrow#.z.N;nrow?syms;nrow?3?1.0*1+til 10;nrow?10*1+til 10))
 }

.z.ts:{[]
  publishtrade[]
 }

\t 2000
```

Once the tickerplant and feedhandler processes are up and running the C subscriber process can connect to the tickerplant to receive the data.

5.9.2 Extracting the table schema

Subscriber processes are required to make an initial subscribe request to the tickerplant in order to receive data, see https://code.kx.com/q/cookbook/publish-subscribe/.
This request involves calling the `.u.sub` function with two parameter arguments. The first argument is

the table name and the second are the list of symbols to subscribe to. Specifying a backtick character for either parameter of `.u.sub` means "all", as in, all tables and/or all symbols. If the `.u.sub` function is called synchronously the tickerplant will return the table schema. The following program demonstrates how the initial subscribe request can be made and the column names of table schema extracted. In the case below, a request is made for all trade records.

```
/* File name: schema.c */
#include
"common.h" int
main() {
  J i;
  I handle;
  I portnumber= 5010;
  S hostname= "localhost";
  S usernamePassword=
  "kdb:pass"; K response,
  table, columnNames;

  handle= khpu(hostname, portnumber,
  usernamePassword); if(!handleOk(handle))
    return EXIT_FAILURE;
  response= k(handle, ".u.sub[`trade;`]", (K)
  0); if(isRemoteErr(response)) {
    kclose(handle);
    return
    EXIT_FAILURE;
  }
  // .u.sub returns a two element list
  // containing the table name and schema
  // q)h:hopen 5010
  // q)h".u.sub[`trade;`]"
  // `trade
  // +`time`sym`price`size!((();();();()))
  if(response->t != 0 || response->n != 2 || kK(response)[0]->t != -
     KS || kK(response)[1]->t != XT) {
    fprintf(stderr,
            "Subscription response is of unknown shape. Top level type is
            %d\n", response->t);
    r0(response);
    kclose(handle);
    return
    EXIT_FAILURE;
  }
  printf("Number of elements returned is %lld\n", response-
  >n); printf("Table name: %s\n", kK(response)[0]->s);

  table=
  kK(response)[1]->k;
  columnNames=
  kK(table)[0];

  printf("Num colNames: %lld\n", columnNames-
  >n); for(i= 0; i < columnNames->n; i++) {
    printf("Column %lld is named %s\n", i, kS(columnNames)[i]);
  }

  r0(response);
  kclose(handle);
  return
  EXIT_SUCCESS;
}
```

Compile and Run

```
$gcc  -DKXVER=3 -o  schema schema.c  c.o  -lpthread
$./schema
Number  of  elements  returned  is
2 First element is the table name :
trade Num columnNames :4
Column   0 is  named  time
Column   1 is  named  sym
Column   2 is  named  price
Column   3 is  named  size
```

5.9.3 Subscribing to a tickerplant data feed

Having received the table schema in the previous step, the next action is to listen for incoming data from the tickerplant. In the subsequent code snippet, after the initial subscription request is made, the program enters a repeating while loop in which the k function is called to check if data has been received. Once received the data can be processed. In this example, the processing function is simply to print the incoming trade records to stdout.

```
/* File name: subscriber.c */
  #include "common.h"
  #include "time.h"

  void printTime(J t) {
    time_t timval= t / 1000000000;
    struct tm *timeInfo= localtime(&timval);

    printf("%dD%02d:%02d:%02d.%09lld ", timeInfo->tm_yday, timeInfo-
           >tm_hour, timeInfo->tm_min, timeInfo->tm_sec, t % 1000000000);
  }

  I shapeOfTrade(K x, K
    tableName) { K columns;
    // check that we received 3 element list
    (`upd;`trade;table) if(x->t != 0 || x->n != 3)
      return 0;
    // check that second element is a table name
    if(kK(x)[1]->t != -KS || kK(x)[1]->s !=
    tableName->s)
      return 0;
    // check if last element is a
    table if(kK(x)[2]->t != XT)
      return 0;
    // check that number of
    columns>=4 columns=
    kK(kK(x)[2]->k)[0];
    if(columns->n < 4)
      return 0;
    // you can add more checks here to ensure that types are as expected
    // likely trade
    update return 1;
  }

  int
    main()
    { J i;
    I handle;
    I portnumber= 5010;
    S hostname= "localhost";
    S usernamePassword= "kdb:pass";
    K response, table, tableName, columnNames, columnValues;

    handle= khpu(hostname, portnumber,
    usernamePassword); if(!handleOk(handle))
      return
    EXIT_FAILURE;
    tableName=
    ks("trade");
    response= k(handle, ".u.sub", r1(tableName), ks(""), (K)
    0); if(isRemoteErr(response)) {
      r0(tableName);
      kclose(handle);
      return
      EXIT_FAILURE;
    }
    r0(respon
    se);
    while(1)
    {
      response= k(handle, (S)
      0); if(!response)
        break;

      if(shapeOfTrade(response,
        tableName)) { table=
        kK(response)[2]->k; columnNames=
        kK(table)[0]; columnValues=
        kK(table)[1];
```

```
      printf(" %s %s %s %s\n", kS(columnNames)[0],
             kS(columnNames)[1], kS(columnNames)[2],
             kS(columnNames)[3]);
      printf(" ----------------------------\n");

      for(i= 0; i < kK(columnValues)[0]->n;
        i++) {
        printTime(kJ(kK(columnValues)[0])[i]);
        printf("%s ",
        kS(kK(columnValues)[1])[i]);
        printf("%lf ", kF(kK(columnValues)[2])[i]);
        printf("%lld \n", kJ(kK(columnValues)[3])[i]);
      }
    }
    r0(response);
  }
  r0(tableName);
  kclose(handle);
  return
  EXIT_SUCCESS;
}
```

```
Compile and Run

$gcc -DKXVER=3 -o subscriber subscriber.c c.o -lpthread
$./subscriber
time                 sym price    size
--------------------------------------
0D20:43:58.044653000 APL 4.000000 20
0D20:44:00.045138000 IBM 7.000000 20
0D20:44:00.045138000 VOD 1.000000 90
time                 sym price    size
--------------------------------------
0D20:44:04.044732000 VOD 1.000000 90
0D20:44:04.044732000 APL 1.000000 100
```

5.10 Publishing to a kdb+ tickerplant

As mentioned earlier, kdb+ tickerplants are designed to handle data feeds from publishing processes. Some examples of data publishersinclude

- Market data feedhandler processes for Financial data and algorithmic trading
- Smart Sensor data feeds from Internet of Things (IoT) devices
- Satellite imaging data for geospatial analytics

The next example will demonstrate the steps required to publish data to a kdb+ tickerplant, continuing with the use case of market trade data. To do so, the publisher will use the k function as described previously to send trade records. The k function will be passed four arguments.

1. Handle value - The integer value returned by the khpu function when a socket connection is established.
2. Update function name (.u.upd) - The function executed on the tickerplant which enables the data insertion. It may be called synchronously or asynchronously and takes two arguments, (3) and (4)
3. Table name - The name of the table the data should be inserted into. Table name should be passed as a symbol.
4. Data - The data which will form the rows to be appended to the table. This argument is typically passed as a mixed list object.

```
I  handle;
K  mixedList;
K  tableName  =  ks((S)"tablename");
k(handle,".u.upd",r1(tableName),mixedList, (K)0)
// To continue using the K object, tableName, after the k call, increment its reference count
```

5.10.1 Publishing a single row using a mixed list object

The next example shows how a `K` object, containing a mixed list corresponding to one row of data, can be passed to the `.u.upd` function. Below the `knk` function is used to create the mixed list containing a symbol, float and integer, constituting a single row. The function `.u.upd` is called in the `k` function with two parameters being passes, a symbol object corresponding to the table name and the singleRow object.

```
/* File name: singleRow.c */
#include "common.h"

int main() {
  I handle;
  I portnumber= 5010;
  S hostname= "localhost";
  S usernamePassword= "kdb:pass";
  K result, singleRow;

  handle= khpu(hostname, portnumber, usernamePassword);
  if(!handleOk(handle))
    return EXIT_FAILURE;

  singleRow= knk(3, ks((S) "ABC"), kf(10.0), kj(20));

  // Perform single row insert, tickerplant will add timestamp column itself

  result= k(handle, ".u.upd", ks((S) "trade"), singleRow, (K) 0);
  if(isRemoteErr(result)) {
    kclose(handle);
    return EXIT_FAILURE;
  }

  r0(result);
  kclose(handle);
  return EXIT_SUCCESS;
}
```

5.10.2 Publishing multiple rows using a mixed list object

For performance related reasons, previously highlighted in the following whitepaper, http://code.kx.com/q/wp/kdbtick_profiling_for_throughput_optimization.pdf, it is recommended to publish multiple rows at once so that bulk inserts can be performed on the tickerplant, maximizing tickerplant efficiency.

"Feeds should read as many messages off the socket as possible and send bulk updates to the tickerplant if possible. Bulk updates will greatly increase the maximum throughput achievable"

The next example demonstrates how such bulk inserts can be performed using a single `k` function call. Again the `knk` function is used to create a mixed list object, however this time each element in the list is itself a simple list object.

```
/* File name: multiRow.c */
#include "common.h"

int main()
  { int i,
  n= 3; I
  handle;
  I portnumber= 5010;
  S hostname= "localhost";
  S usernamePassword= "kdb:pass";
  S symbols[]= { "ABC", "DEF", "GHI" };
  K result;

  handle= khpu(hostname, portnumber, usernamePassword);
  if(!handleOk(handle))
    return EXIT_FAILURE;

  K multipleRow= knk(3, ktn(KS, n), ktn(KF, n), ktn(KJ, n));
  for(i= 0; i < n; i++) {
    kS(kK(multipleRow)[0])[i]= ss(symbols[i % n]);
    kF(kK(multipleRow)[1])[i]= 10.0 * i;
    kJ(kK(multipleRow)[2])[i]= i;
  }

  // Perform multiple row insert, tickerplant will add timestamp column
  itself result= k(handle, ".u.upd", ks((S) "trade"), multipleRow, (K) 0);
  if(isRemoteErr(result)) {
    kclose(handle);
    return
    EXIT_FAILURE;
  }

  r0(result);
  kclose(handle);
  return
  EXIT_SUCCESS;
}
```

5.10.3 Adding a timespan column

In the previous example, the tickerplant added a timespan column to each row received because one was not already present. This is standard tickerplant functionality and is used to record when the data was received by the tickerplant. Instead, it is possible for the publisher to create the timespan column to prevent the tickerplant from adding one. In the example below it is shown how the previous code can be extended to include a timespan column in the mixedList object sent to the tickerplant.

```
/* File name: rowswithtime.c */
#include <time.h>
#include "common.h"

J castTime(struct tm *x) {
  return (J)((60 * x->tm_hour + x->tm_min) * 60 + x->tm_sec) * 1000000000;
}

int
  main()
  { J i,
  n= 3;
  I handle;
  I portnumber= 5010;
  S hostname=
  "localhost"; K
  result;
  S usernamePassword= "kdb:pass";
  S symbols[]= { "ABC", "DEF", "GHI" };
  time_t
  currentTime;
  struct tm *ct;

  handle= khpu(hostname, portnumber, usernamePassword);
  if(!handleOk(handle))
    return EXIT_FAILURE;

  K multipleRow= knk(4, ktn(KN, n), ktn(KS, n), ktn(KF, n), ktn(KJ, n));
  time(&currentTime);
  ct= localtime(&currentTime);

  for(i= 0; i < n; i++) {
    kJ(kK(multipleRow)[0])[i]= castTime(ct);
    kS(kK(multipleRow)[1])[i]= ss(symbols[i %
    n]);
    kF(kK(multipleRow)[2])[i]= 10.0 * i;
    kJ(kK(multipleRow)[3])[i]= i;
  }

  result= k(handle, ".u.upd", ks((S) "trade"), multipleRow, (K) 0);
  if(isRemoteErr(result)) {
    kclose(handle);
    return
    EXIT_FAILURE;
  }

  r0(result);
  kclose(handle);
  return
  EXIT_SUCCESS;
}
```

5.11 Conclusion

This document covered multiple aspects of the C API interface for connecting with the Kx systems kdb+ database. Topics covered included the creation of socket connections, execution of queries, error handling, memory management, and the creation and extraction of data from K objects such as lists, dictionaries and tables. Practical examples formed the basis for the construction of a C subscriber process, capable of consuming a kdb+ data feed, and a feedhandler process designed to publish data to a kdb+ tickerplant. Finally, the use of shared C libraries to extend the functionality of kdb+ was also demonstrated.

it's about time

Technical Whitepaper

6 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.

CONTENTS

6.1 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.

6.2 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+

6.2.1 Monadic functions

6.2.1.1 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
```

6.2.1.2 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
```

6.2.1.3 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
```

6.2.1.4 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
```

6.2.2 Dyadic functions

6.2.2.1 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
```

6.2.2.2 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
```

6.2.2.3 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
```

6.2.2.4 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"
```

6.2.2.5 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
```

6.2.2.6 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
```

6.2.3 Higher valence functions

6.2.3.1 Each-Both

```
q) 1 2 3 in' (1 2 3;3 4 5;5 6 7)
110b
```

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.

6.2.3.2 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
```

6.3 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)
```

6.4 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]
```

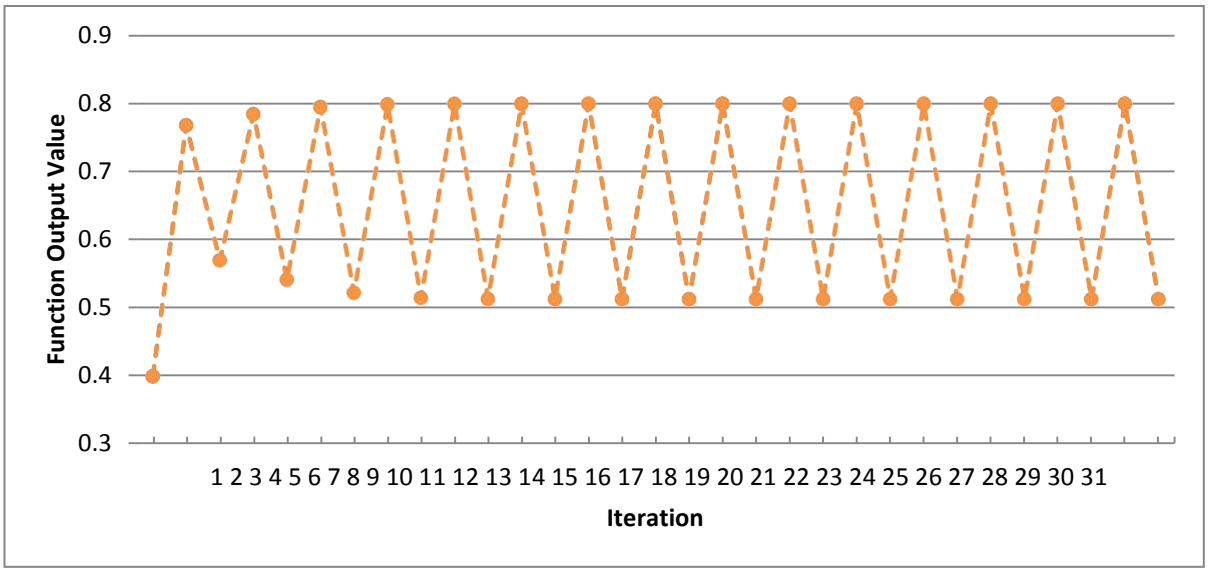


Figure 1: Infinite looping function example

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.

6.5 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
```

6.6 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
```

6.7 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)

it's about time

Technical Whitepaper

7 Multi-threading in kdb+: Performance Optimizations and Use Cases

Author:

Edward Cormack is a Financial Engineer who has worked on developing financial data solutions for a range of clients. Edward is currently based at a top-tier investment bank in London, developing a kdb+ platform for global equity risk.

CONTENTS

7.1 INTRODUCTION

Parallel processing techniques have become increasingly prevalent in mainstream computing as a result of developments in computer architecture. Multi-core processors are now widely used in all systems from mobile devices to high-performance servers.

With this comes the need to develop applications that can make full use of parallel architecture to realise performance gains. Parallel computing techniques typically involve dividing large tasks into several parts, which can be executed simultaneously, taking less time than it would take to run all the parts sequentially. This concurrent operation shares the load, using the full resources of the system to optimise performance.

This whitepaper intends to introduce the methods available for parallel processing in kdb+. We will demonstrate a number of cases where parallel techniques can be employed and discuss the potential performance enhancements.

In many cases, operations on large vector datasets, which are typical of those processed and analysed by kdb+, are easily parallelised. Where this is the case, complex operations on long lists can be broken up and executed on multiple threads.

We will also address use cases where parallel processing may not be appropriate in kdb+ and some of the important factors to consider in each case.

The OS disk cache is flushed when required using the io.q script which can be obtained from the Kx code repository (http://code.kx.com/wsvn/code/contrib/simon/io/io.q).

All tests performed using kdb+ version 3.1 (2013.06.25)

7.2 SETTING UP Q FOR PARALLEL PROCESSING

Kdb+ starts in single threaded mode by default. This ensures data consistency and that there are no race conditions as all commands are executed in the order they are received. Multi-threading in kdb+ is achieved by spawning multiple slave threads.

To perform parallel operations in kdb+, the q process is started up using the `-s n` option. When called with a value of n>1, kdb+ will start n slave threads. For optimal performance when using multi-threading, the number of slaves should be equal to or greater than the number of available cores. One should take into consideration the overall usage of the host when determining the number of slaves to allocate.

Parallel execution is explicitly invoked by using two built-in functions: `peach` and `.Q.fc`. Parallel execution is also implicitly invoked in kdb+ when used as part of a multi-threaded HDB. This mode of operation will be reviewed in more detail in section 4.

7.2.1 Functions for parallel execution

`peach`, or parallel-each is used in the same way as the `each` adverb. It will execute the function over multiple slaves, passing the arguments and results between the slaves and the main thread using IPC serialisation.

```
q) f peach x      // execute function f on x over slave threads
```

Additionally, there is the parallel cut function `.Q.fc`. This modifies an existing function such that a vector argument is distributed across multiple threads. The differences between `peach` and `.Q.fc` and the various use cases are covered in more detail in section 3.

```
q) .Q.fc[f] z    // cuts z into n parts and executes f in parallel
```

There are however, some limitations on what operations can be executed in parallel. In order to maintain thread safety, q utilises thread-local storage. Global variables may be read directly from the main thread but each thread maintains its own copy of local variables and arguments.

To avoid concurrency issues, only the main thread may update global variables. Attempting to set or modify a global while in a slave thread will result in a `'noupdate` error. There are also restrictions on calling system commands while using `peach`.

```
q) { update ex:"N" from `trade where sym=x } peach `GOOG`AAPL`YHOO
'noupdate
```

7.3 VECTOR OPERATIONS IN PARALLEL

In this section, we will look in more detail at the use of `peach` and `.Q.fc` for operations on vectors. We analyse the performance of a range of use cases using `\t` to time execution on a process with 6 slaves for both sequential and parallel operations.

Consider the lambda demonstrated below. This will create a list of floats of the same length as the argument and raise it to a specific power, then return the sum of the result.

```
// Run operation 6 times, for a 1 million element list on each iteration

q)\s
6
q)\t { sum (x?1.0) xexp 1.7 } each 6#1000000
802
q)\t { sum (x?1.0) xexp 1.7 } peach 6#1000000
136
```

The advantage of using `peach` in this case is clear. Distributing the operation over all 6 threads increases the speed by a factor of approximately 5.9.

Data and results are copied between slaves and the main thread using IPC serialisation. As such, there is an associated overhead cost with transferring data which should be taken into account. If we modify the existing function so that it returns the entire vector instead of the sum then we can see this effect.

```
// Returning the full result is worse as the overhead is increased

q)\t { (x?1.0) xexp 1.7 } each 6#1000000
795
q)\t { (x?1.0) xexp 1.7 } peach 6#1000000
152
```

Now, the single threaded operation is slightly faster than before as it does not need to compute the sum. The parallel operation must now return a 1 million element vector to the main thread, and it is slower as a result. The performance scaling is now reduced to 5.2.

The parallel overhead from copying data to slave threads can be estimated by using the `-8!` operator to serialise the argument and then de-serialising with the `-9!` operator.

```
q)\t:100 -9!-8!1000000?1.0
732
```

The operation must be computationally expensive enough to justify this messaging overhead. In the case of a very simple operation, using `peach` can be slower than just executing sequentially.

```
// Peach takes longer when just doing a product

q)\t:1000 { x*1?10 } each 10?1.0
3
q)\t:1000 { x*1?10 } peach 10?1.0
18

// Increasing the complexity makes the overhead worthwhile

q)\t:100 {  sqrt (1000?x) xexp 1.7 } each 10?1.0
145
q)\t:100 {  sqrt (1000?x) xexp 1.7 } peach 10?1.0
33
```

7.3.1 Vector operations and parallel cut

Just as when operating in single-threaded mode, operations on vectors are generally more efficient. Where appropriate, it is preferable to call a single function with one long vector argument than to call the function many times; i.e. iterating over a list of atomic arguments.

```
// Single-threaded mode. Atomic vs. vector argument

q)a:1000000?1.0

// Taking the exponent of each value and sum the result

q)sum { x xexp 1.7 } each a
370110f
q)\t sum { x xexp 1.7 } each a
321

// Apply sum and exponent operations to the whole vector instead

q){ sum x xexp 1.7 } a
370110f
q)\t { sum x xexp 1.7 } a
132
```

To carry out vector operations in parallel we use `.Q.fc`, or parallel cut. As the name suggests, this will break a vector argument into n parts, and pass each part to a slave thread for execution.

In the worst case example below, `peach` is used over a long list of atomic values so that each value must be passed to the slave thread and the value returned. This maximises the data transfer overhead. Using `.Q.fc` is much more efficient as the function is only executed 6 times – once on each thread.

```
// Performance of peach is slower than single-thread each
q)\t { sum x xexp 1.7 } peach a
448

// Parallel speed-up is observed using .Q.fc
q)\t .Q.fc[{ sum x xexp 1.7 }] a
28
```

7.3.2 Workload balancing

Another factor to take into consideration when using multi-threading is the distribution of work across the threads. This is particularly important when the process is handling a large number of tasks of uneven size.

The way in which q splits work between threads depends on which mode of parallel execution is used. Both parallel functions use pre-assignment, but with slightly different behaviour. When using `peach` over 2 slaves the first slave will get arguments 0, 2, 4… and the second slave gets arguments at position 1, 3, 5… etc.

This allocation can be shown in q using the number of slaves and the modulus function, `mod`.

```
// 6 input arguments over 2 cores
q)(til 6) mod 2
0 1 0 1 0 1

// 14 input arguments over 6 cores
q)(til 14) mod 6
0 1 2 3 4 5 0 1 2 3 4 5 0 1
```

As discussed earlier, `.Q.fc` distributes work by cutting the vector argument into n slices. Each thread will then execute one slice of the original list.

To demonstrate the effect of uneven distribution we define a function f with an execution time proportional to the input argument.

```
q)f: { x:7000*x;sum sqrt (x?1.0) xexp 1.7;};
q)\t f[100]
102
q)\t f[1000]
1053
```

A q process is now started up with 2 slaves; this makes it clearer to follow the operation. The test function is executed over a list of varying input arguments. We demonstrate with the examples below that the ordering of the large and small arguments can have a significant effect on performance.

```
// Single threaded
q)\t f each 10 10 10 10 1000 1000 1000 1000
4162
// Single threaded unbalanced
q)\t f each 10 1000 10 1000 10 1000 10 1000
4163
// Multi threaded unbalanced
q)\t f peach 10 1000 10 1000 10 1000 10 1000
4131
// Multi threaded balanced
q)\t f peach 10 10 10 10 1000 1000 1000 1000
2090
```

Executing in single-threaded mode, we find the total time is similar to the sum of the arguments, as we would expect. In the second case however, there is little to no improvement from using `peach`. This is because the fast and slow jobs alternate in the input list, which means all the slow tasks are assigned to a single thread. The main process must wait for the second thread to execute all 4 slow jobs even though the first thread is finished. The final case is a balanced distribution, with fast and slow jobs assigned evenly to each thread. Now we see a parallel speed-up factor of approximately 2 as expected.

For `.Q.fc`, the situation is reversed. This function splits a vector argument evenly along its length and assigns one slice to each slave. Alternating values in the list will result in balanced threads but contiguous blocks of large or small values will not.

```
// Unbalanced – first 4 fast arguments sent to first thread
q)\t .Q.fc[{f each x}] 10 10 10 10 1000 1000 1000 1000
4134

// Balanced distribution
q)\t .Q.fc[{f each x}] 10 1000 10 1000 10 1000 10 1000
2112
```

This pre-assignment can be overridden by passing a nested list to `peach`. If you wanted to send specific arguments to each thread, then these can be positioned in the input with the index of the top-level list corresponding to the thread number.

For example, the following input will execute values 0, 1, 4 from x on the first thread and 2, 3, 5 on the second.

```
q)x
10 10 1000 1000 10 1000
q)\t f peach x[0 1 4], x[2 3 5]
2075
```

7.4 QUERY OPTIMISATIONS USING MULTI-THREADING

The partitioned database structure in kdb+ is well suited to parallel processing. In a standard date partitioned DB, the data is arranged such that all data for a given day resides in a specific directory under the root. The advantage of this is that the q process is only required to read data from the partitions specified in the date constraint.

This ability to access sections of the database independently can be extended using slaves, with each slave being assigned a date from the `where` clause to process. Considering the following query;

```
q) { select from trade where date = x } peach d
```

We can see how `peach` will assign each date in the list `d` to a slave thread for processing. In practice, q handles multi-threaded HDB queries under the covers without the need for any additional functions. It will automatically distribute work across slaves and aggregate the results back to the main thread.

This section discusses various applications for the multi-threaded HDB and what performance improvements can be achieved.

7.4.1 Overview of test data

The test database is a standard TAQ schema containing simulated data for the S&P 500 components for one month partitioned by date. There are approximately 20,000 quotes per tick symbol per day.

The data is sorted by symbol and time. The parted attribute is applied on sym.

```
q)meta quote
c    | t f a
-----| -----
date | d
sym  | s   p
time | t
bid  | e
ask  | e
bsize| j
asize| j
ex   | c

q)meta trade
c    | t f a
-----| -----
date | d
sym  | s   p
time | t
price| e
size | j
ex   | c
```

The use of attributes to optimize datasets has been reviewed in an earlier paper from this series; March 2013 : Columnar Database and Query Optimization.

In each of the test cases outlined in this section, the q session is restarted between each test and the disk cache flushed using io.q.

7.4.2 Performance effects of standard queries with slaves

If we consider a normal select from the quote table, for a range of symbols;

```
q) select from quote where date in d, sym in `GOOG`AAPL`YHOO`AMZN`EBAY
```

This query will return approximately 100,000 rows per day. The query is executed for an increasing number of days, and the performance measured. The test is then repeated on a process using additional slave threads for comparison. The results are shown in Table 1 below.

	0 slaves	6 slaves	12 slaves
Days in query	Execution time (ms)		
5	195.8	162.5	160.5
10	384.6	328	317.9
15	595.6	507.5	477.3
20	839.8	677	624.1

Table 1: Performance of normal select with increasing days

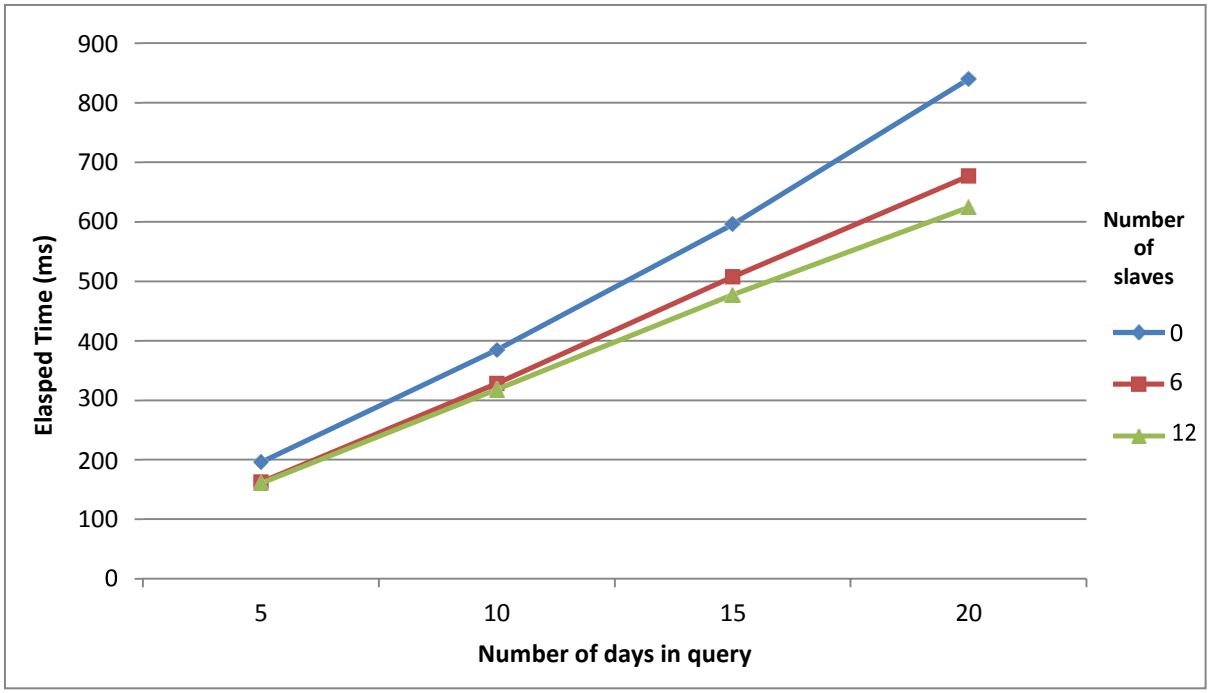


Figure 1: Effect of number of slave threads on query time

The results in Fig.1 show a gradual increase in performance as we add more slave threads. The relation between execution time and query time remains roughly linear; however the factor by which it increases is reduced.

7.4.3 Map-reduce with multi-threading

One area in which there are significant advantages to using parallel operation is when we are performing aggregations over a partitioned HDB. For most common aggregation operations, kdb+ has a built in map-reduce feature.

For example, if we wanted to calculate an average value over multiple days then we can calculate the sum and count for each partition. We then take the total sum over the total row count to reduce to a final value. This saves the overhead of loading the data from all partitions into memory to perform the calculation.

Map-reduce can be used for the following operations: `avg`, `cor`, `count`, `cov`, `dev`, `first`, `last`, `max`, `min`, `prd`, `sum`, `var`, `wavg` and `wsum`. If kdb+ detects that a compatible operation is being used in the query then it will map the query over the partitions.

The advantage of map-reduce can be seen by comparing the following queries which calculate a VWAP over multiple days.

```
// Standard query form using map-reduce
q)t1:select vwap: size wavg price by sym from trade where date in 5#date

// Non map-reduced form of the same query
q)t2:select vwap:(sum size*price)%sum size by sym from select
sym,size,price from trade where date in 5#date

// Both queries return the same result
q)t1 ~ t2
1b
```

The second query is written such that it will delay the aggregation step until after data has been loaded from all partitions. This will not be optimised by map-reduce.

	Map-reduce	Non map-reduce
Days in query	Elapsed time (ms)	
5	692.4	963.4
10	1370.2	1923.8
15	2053	2906.6
20	2726.6	4085.6

Table 2: Comparison of using map-reduce for aggregations

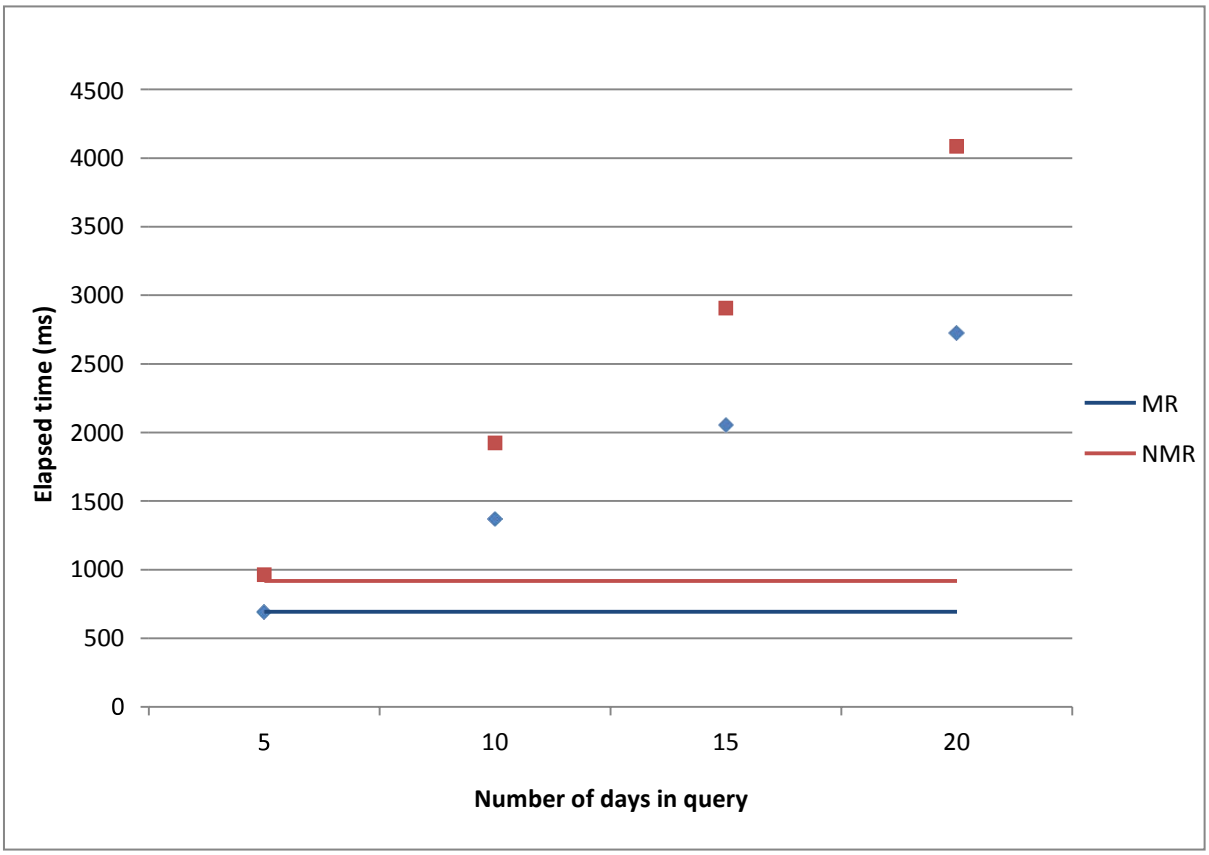


Figure 2: Performance increases from using map-reduce

It can be seen in Figure 2 that there is a significant slowdown when not using map-reduce operations. The execution time increases more rapidly with query size than the map-reduced version.

The multi-day VWAP query above is now repeated using slaves. In this case, kdb+ will distribute the map calculations to its slaves. Again, this functionality is automatic and requires no specific functions to be called.

	0 slaves	6 slaves	12 slaves
Days in query	Elapsed time (ms)		
5	692.4	233.6	221.2
10	1370.2	477.6	477.8
15	2053	720.8	737.6
20	2726.6	954.6	898.2

Table 3: Map-reduce performance using slaves

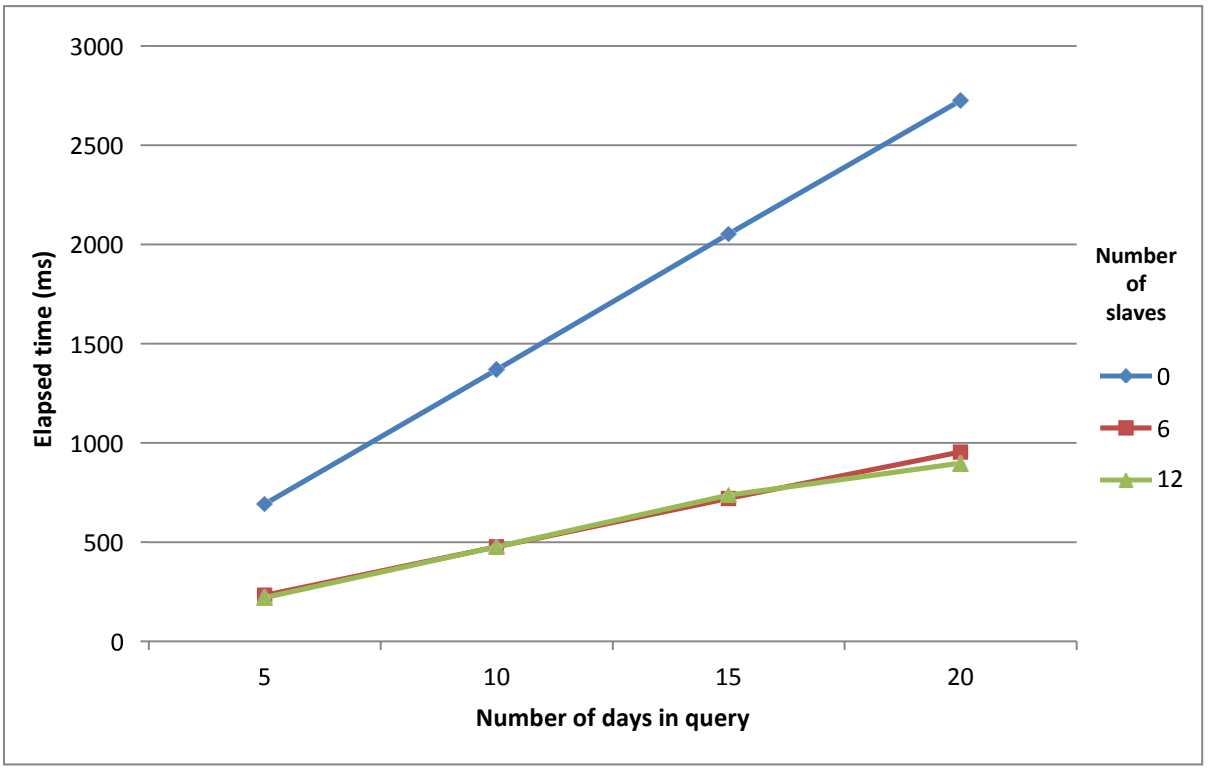


Figure 3: Execution time of multi-threaded map-reduce queries with date range

Figure 3 shows that there is a significant increase in the performance of the map-reduce query when used in multi-threaded mode. This is in line with our expectations, as the calculations can be spread across slaves and executed in parallel.

It is notable that there is little increase in performance from using 12 slaves instead of 6. A slight increase can be observed for the maximum number of dates in this sample database, although we can extrapolate that a greater number of slaves will perform better as the size of the query increases. The likely explanation for this behaviour is I/O saturation. Increasing the number of slaves available for calculation does not improve performance significantly as reading the data from disk becomes the limiting factor. Adding additional slaves will only increase the number of processes attempting to read from the disk simultaneously.

NOTE: This result is for demonstration purposes only, and the actual usage of parallel queries will depend on the specific system set-up of each user. Thorough testing across a range of database parameters is recommended before implementing any parallel query arrangement.

7.4.4 Parallel access and segmentation

kdb+ supports parallel I/O to address the disk contention issue outlined earlier. This requires a specific hardware arrangement, usually direct attached storage, with multiple disks residing on separate I/O channels.

To allow simultaneous access across partitions the database structure is modified to use segments. Segmentation adds an additional layer above the partitioned structure. A simple example of this structure would be to split a date partitioned database across two disks, by saving alternating days on each segment.

When using a segmented database for parallel queries, it is recommended to have at least one slave thread per disk array. This will allow maximum access for date range queries. In the case of computationally expensive queries, e.g. calculating VWAP, then 2 threads can be assigned per core. In this case, one thread is performing the calculations while another is loading the data.

A fully multi-threaded segmented database requires a custom hardware set up and is generally tailored to the needs of a specific system. As such, benchmarking the performance of this sort of database is beyond the scope of this paper.

7.5 FILE OPERATIONS

7.5.1 DB maintenance

Adding or modifying a column to a partitioned database can be slow when carried out on large tables. This operation can be improved using `peach`. A shorter execution time for these maintenance operations can help reduce the duration required to make changes to production systems.

In the example below, we have added a column to the quote table on the test database. This is done using the functions in the dbmaint.q script;

(http://code.kx.com/wsvn/code/contrib/fdsupport/database%20maintenance/dbmaint.q)

```
q)\t addcol[hsym `$"/tmp/db"; `quote; `newcol; 0N]
2013.05.26 14:26:12 adding column newcol (type -7) to
`:/tmp/db/2013.07.01/quote ...
4388
q)deletecol[hsym `$"/tmp/db"; `quote; `newcol]
2013.05.26 14:26:33 deleting column newcol from
`:/tmp/db/2013.07.01/quote ...
```

The standard `addcol` operation takes just over 4 seconds to run across one month of test data. The new column is then removed, and then test repeated using a modified version, which updates all partitions in parallel: `p_addcol`.

```
// Define p_addcol using peach

q)p_addcol: {[dbdir;table;colname;defaultvalue]
if[not validcolname colname;'(`)sv colname,`invalid.colname];
add1col[;colname;enum[dbdir;defaultvalue]] peach allpaths[dbdir;table];}

q)\s
6
q)\t p_addcol[hsym `$"/tmp/db"; `quote; `newcol; 0N]
2013.05.26 14:27:26 adding column newcol (type -7) to
`:/tmp/db/2013.07.06/quote ...
1743
```

The disk maintenance operation is now measured to take less than half the time when executing in parallel.

7.5.2 Writing flat files

Reading and writing data from disk, particularly in comma or tab delimited format can be a time consuming operation. We can use slaves to speed up this process, particularly when we need to write multiple files from one process.

For example, we may wish to save the result of a kdb+ query which spans several days in CSV format, creating a separate file for each date. Alternatively, there may be a number of flat files which we want to load into a kdb+ session.

The example below uses an extract from the trade table on the test database. The table is cut into 5 slices and each slice is saved to disk with a separate filename.

```
q)\s
6
q)td: select from trade where date=2013.05.01, sym in 5?sym
q)count td
49815
q)tds: (floor (count td)%5) cut td
q)\t {(hsym `$"/tmp/testfile",string 1+x) 0: csv 0: tds[x]} each til 5
75
q)\t {(hsym `$"/tmp/testfile",string 1+x) 0: csv 0: tds[x]} peach til 5
29

q)count each tds
9963 9963 9963 9963 9963
q)system "ls /tmp | grep test*"
"testfile1"
"testfile2"
"testfile3"
"testfile4"
"testfile5"
```

The second query, using `peach` is measured to take less than half the time to execute. The reverse operation is then demonstrated, reading all of the files back into the q session.

```
q)read0 `$"/tmp/testfile1"
"date,sym,time,price,size,ex"
"2013-07-01,GOOG,09:30:00.069,1.657921,2,A"
"2013-07-01,EBAY,09:30:00.232,1.463019,3,A"
"2013-07-01,AAPL,09:30:00.266,0.569119,3,N"
"2013-07-01,EBAY,09:30:00.333,1.681702,2,N"
...
```

```
q)\t raze {flip (cols trade)!1_/:("DSTFIC";csv) 0: (hsym
`$"/tmp/testfile",string 1+x) } each til 5
50
q)\t raze {flip (cols trade)!1_/:("DSTFIC";csv) 0: (hsym
`$"/tmp/testfile",string 1+x) } peach til 5
17

q)td: raze {flip (cols trade)!1_/:("DSTFIC";csv) 0: (hsym
`$"/tmp/testfile",string 1+x) } peach til 5
q)count td
49815
q)meta td
c    | t f a
-----| -----
date | d
sym  | s
time | t
price| f
size | i
ex | c
```

Again, the parallel operation is more than twice as fast at reading the data. Combining all the results together we get the same schema and row count as the original data.

7.6 CONCLUSION

This paper has summarised a range of applications for multi-threading in kdb+. These included optimisation of queries, parallel execution of algorithms, vector operations and file operations. The mechanism behind slave threads has been introduced and the relevant functions explained.

For each of these cases, the potential improvements have been measured and any limitations or potential problems identified. While there is a lot to be gained from using parallel operation appropriately, it is far from a catch-all solution to performance problems. Prior to attempting parallelisation, operations should be fully vectorised to ensure maximum benefit.

Used incorrectly, parallel processing can be less efficient than the equivalent single-threaded operation due to the overheads of transferring data between threads or unbalanced workloads. Also, some operations are unsuitable for multi-threading, as they may require access to global variables.

The use of multi-threading in kdb+ should therefore be treated on a case by case basis. It is a feature which should often be considered by developers, but performance testing is recommended before implementation.

System management solutions are available to assist with administering multi-process environments, like Kx for Control which provides a range of tools for visualising process workflow, task scheduling and system resource monitoring. These features are especially helpful at managing master/slave type configurations, where client queries or the loading of data is divided between processes in order to increase throughput.

All tests performed using kdb+ version 3.1 (2013.06.25)

it's about time

Technical Whitepaper

8 Kdb+ tick Profiling for Throughput Optimization

Author:

Ian Kilpatrick has worked on several kdb+ systems. Based in Belfast, Ian is a Technical Architect for the Kx suite of high performance data management, event processing and trading platforms.

CONTENTS

8.1 INTRODUCTION

kdb+ is seen as the technology of choice for many of the world's top financial institutions when implementing a tick capture system. kdb+ is capable of processing large amounts of data in a very short space of time, making it the ideal technology for dealing with the ever-increasing volumes of financial tick data. The core of a kdb+ tick capture system is the tickerplant. Kx have made available the source code for theirproduct kdb+tick which will form the basis of this paper. The purpose of this whitepaperis to discuss factors which influence messaging and throughput performance fora kdb+ based tick capture system and to present a methodology with which this performance can be profiled to assist in optimizing the tick system configuration.

Some of the possible factors are:

- Number of rows in each update
- Size of the data in bytes
- Tickerplant publish frequency
- Number of subscribers

- Network latency and bandwidth
- Disk write speed
- TCP/IP tuning
- Version of kdb+

This paper examines the first 4 of these. All tests were performed on 64 bit Linux with 8 CPUs, using kdb+ version 3.1 (2014.02.08).

For more information on kdb+ tickerplants, see http://code.kx.com/wiki/Startingkdbplus/tick.

NOTE: The results presented in this whitepaper are for indication purposes only. Results will vary for each individual kdb+ system, especially when hardware specifications are taken into consideration.

8.2 SETUP

We will run a feed simulator which will publish trade data to a tickerplant (TP) on a timer. The trade table has the following schema:

```
trade:([]time:"P"$();sym:`g#"S"$();price:"F"$();size:"I"$();cond:())
```

We will also run an RDB (real-timedatabase) which subscribes to the tickerplant fortrade messages and inserts into an in-memory table. All the processes will run on the same server and we will use `taskset` to tie each process to a different CPU.

e.g.

```
taskset –c 0 q tp.q
taskset –c 1 q rdb.q
taskset –c 2 q
feedsim.q
```

8.2.1 Feed simulator code

```
/ connect to
tickerplant h:hopen
8099

/ number of extra columns to add for
test 3.2 ex:0

/ number of rows to send in each
update r:10

/ number of updates to send per
millisecond u:1

/ timer
frequency
t:1

/ timer function, sends data to the tickerplant
.z.ts:{
  data:(r#.z.p;r?`3;100*r?1.0;10*r?100;
  r#" "); if[ex>0; data,:ex#enlist
  r#1f]; if[r=1;data:first each data];
  do[u;neg[h](`upd;`trade;data);neg[h][
  ]];
 }

system"t ",string t
```

8.2.2 Tickerplant code

```
/ listen on port 8099
\p 8099

/ dictionary to contain handles to publish to
subs:enlist[`trade]!();

/ function to subscribe to a table
sub:{[t] subs[t],:neg .z.w;}

/ remove subscriber if connection is lost
.z.pc:{subs::subs except \: neg x;}

/ create empty log file
logFile:`$":sym",string .z.D;
logFile set ();
numMsgs:0;
fileHandle:hopen  logFile;

/ write data to logFile and publish to subscribers, called by the feed sim
upd:{[t;x]
  tm1:.z.p;
  fileHandle@enlist(`upd;t;x);
  numMsgs+:1;
  tplog,:0.001*.z.p-tm1;
  tm2:.z.p;
  subs[t]@\:(`upd;t;x;tm2);
  tppub,:0.001*.z.p-tm2;
 }
```

The tickerplant described above differs from a standard tickerplant in a number of ways. A standard tickerplant would:

- Check to see if the log file already exists on startup and then read the number of messages it contains
- The `sub` function could be enhanced to handle subscribing for certain syms only
- The `upd` function could add a time to the data if it is not present
- The `upd` function would only send `(`upd;t;x)` to the subscribers
- At EOD it would send a message to any subscribers and roll the log file

In the tickerplant described above, we are capturing some extra timing metrics which will allow us to profile the messaging and throughput statistics for the tickerplant. These are described in section 3.

8.2.3 RDB code

```
/ define trade table
trade:([]time:"P"$();sym:`g#"S"$();price:"F"$();size:"I"$();cond:())

/ number of extra columns to add for test 3.2
ex:0
if[ex>0; trade[`$"col",/:string til ex]:ex#enlist"F"$()]

/ define upd to insert to the table, called by tickerplant
upd:{[t;x;tm2]
  tm3:.z.p;
  insert[t;x];
  tm4:.z.p;
  rdbrecv,:0.001*tm3-tm2;
  rdbupd,:0.001*tm4-tm3;
 }

/ connect to tickerplant and subscribe for trades
h:hopen 8099;
h(`sub;`trade);
```

Again, the RDB described here differs from a standard RDB. A standard RDB would:

- Replay the tickerplant log file on startup to get trades from earlier in the day
- upd would only take 2 parameters, the table name and data
- There would be an EOD function defined which would write intra-day tables to disk and then empty the tables

Similarly to the TP code, the code for the RDB records some timing metrics to measure throughput on the RDB. These are described in section 3.

8.3 TESTS

For each test we will vary certain parameters in the feed simulator code and record the median values for `tplog, tppub, rdbrecv, rdbupd` in the code above where

- `tplog` = median time in microseconds for the tickerplant to write the data to the log file
- `tppub` = median time in microseconds for the tickerplant to publish the data to the subscribers
- `rdbrecv` = median time in microseconds for the RDB to receive the message from the tickerplant
- `rdbupd` = median time in microseconds for the RDB to insert the

data We will also record

- Rows per upd = number of rows received by the tickerplant per update
- Rows per sec = number of rows received by the tickerplant per second
- TP CPU = CPU usage of the tickerplant seen using `top` (with the timing code in the `upd` function removed)
- RDB CPU = CPU usage of the RDB seen using `top` (with the timing code in the `upd` function removed)

Testing Specifications

- All tests were run on 64bit Linux with 8 CPUs, using kdb+3.1 2014.02.08.
- The tickerplant is writing to local disk and the write speed is 400MB/s.
- All the processes run on the same server and are run using `taskset`. Note that kdb+ does not implement IPC compression when publishing to localhost.
- Each test is run until the median times stop changing, which takes roughly 1 or 2 minutes.

8.3.1 Number of rowsin each update

In this test we will vary the numberof rows sent in each update by changing the values of `r` (number of rows per update), `u` (numberof updates pertimerfrequency) and `t` (timerfrequency in milliseconds) in the feed simulator code. The number of rows received by the tickerplant per second is calculated as:

Rows per second = `r * u * 1000 / t`

Rows per upd (r)	Upds per timer (u)	Timer freq (t)	Rows per sec	tplog	tppub	TP CPU	rdbrecv	rdbupd	RDB CPU
1	10	1	10,000	14	3	31	71	4	12
10	1	1	10,000	19	4	6	80	10	2
100	1	10	10,000	35	7	1	106	46	1
1	30	1	30,000	13	3	92	80	4	24
10	3	1	30,000	16	4	4	85	7	3
100	3	10	30,000	30	6	1	99	44	1
10	10	1	100,000	15	4	32	82	7	17
100	1	1	100,000	32	6	6	103	46	4
1000	1	10	100,000	121	23	2	224	378	3
100	5	1	500,000	28	6	32	105	42	22

Table 1 – Microseconds taken and percent CPU by rows per update

As seen in Table 1, the results mainly depend on the rows per upd (`r`) parameter and not the other 2 parameters. When publishing the data in single rows (rows per upd = 1) we can only achieve about 30,000 rows persecond before the TP CPU usage approaches 100%. If the data is in 10 rows per update we can handle over 100,000 rows per second.

Significant time is saved when processing 10 rows at a time. In fact, it takes only a little more time than processing 1 row. This goes to the heart of q: in a vector based language bulk operations are more efficient than multiple single operations.

Rows per upd (r)	1	10	100
tplog	13	17	31
tppub	3	4	6
rdbrecv	75	90	103
rdbupd	4	8	44

Table 2 – Microseconds taken by rows per update

From these results we can conclude that feeds should read as many messages off the socket as possible and send bulk updates to the tickerplant if possible.

8.3.2 Size of row in bytes

In this test we will vary the number of columns in the trade table by changing `ex` (number of extra columns) in the feedsim and RDB code.

- The times for ex=0 have already been found in section 3.1.
- For each update message, we determine the size using `count -8!x`.

Num extra cols (ex)	Rows per upd (r)	Rows per sec	tplog	tppub	TP CPU	rdbrecv	rdbupd	RDB CPU	Size of upd in bytes
0	1	10,000	14	3	31	82	4	12	73
10	1	10,000	14	4	34	87	6	16	213
50	1	10,000	17	5	43	79	12	38	773
0	10	100,000	15	4	32	86	7	17	334
10	10	100,000	18	5	40	78	8	19	1194
50	10	100,000	33	7	60	92	14	40	4634
0	100	100,000	28	6	6	106	41	4	2944
10	100	100,000	50	8	13	187	46	8	11004
50	100	100,000	141	16	28	336	61	23	43244

Table 3 – Microseconds taken and percent CPU by size of row

We can see the `tplog`, `tppub` and `rdbupd` times increase as the number of columns increase, as we would expect. The increase is more noticeable for large bulk updates: if there is only 1 row per update then adding 10 columns to the published dataonly increases the CPU usage of the tickerplant by 10%.

8.3.3 Publish frequency

In this test we will alter the tickerplant behaviour so that it publishes to subscribers on a timer. Specifically, we will examine the following three scenarios:

- The tickerplant writes each update to disk individually and publishes each update to subscribers as soon as it is received (standard zero-latency tickerplant)
- The tickerplant writes each update to disk individually but publishes to subscribers on a timer
- The tickerplant both writes to disk and publishes to subscribers on a timer

By batching up updates we can reduce the load on the tickerplant and the RDB. However, this will also result in a delay to RDB receiving the data. Of course, publishing on a timer is not suitable if the subscribers need the data immediately.

We need to make the following adjustments to the tickerplant code in order to publish on a timer:

```
/ define trade table
trade:([]time:"P"$();sym:`g#"S"$();price:"F"$();size:"I"$();cond:())

/ write to disk and insert to the local
table upd:{[t;x]
  tm:.z.p;
  fileHandle@enlist(`up
  d;t;x); numMsgs+:1;
  insert[t;x];
  tpupd,:0.001*.
  z.p-tm;
 }

/ publish the data in the local table and clear
.z.ts:{
  tm:.z.p;
  {[t]
    if[0=count value t; :()];
    subs[t]@\:(`upd;t;value t);
    .[t;();0#];
  } each
  enlist`trade;
  tpflush,:0.001*.
  z.p-tm;
 }
```

To furtherreduce the load we will bufferthe messages and only write to the on-disk log file on a timer as well. However, it should be noted that in the event of a tickerplant going down, more datawill be lost if the data is being logged on a timer than if the data is being written on every update.

In orderto write updates to disk on a timer, we need to make the following changes to the tickerplant code:

```
/ define trade table
trade:([]time:"P"$();sym:`g#"S"$();price:"F"$();size:"I"$();cond:())

/ insert to the local
table upd:{[t;x]
  tm:.z.p;
  insert[t;x];
  tpupd,:0.001
  *.z.p-tm;
 }

/ publish the data in the local table, write to disk and  clear
.z.ts:{
  tm:.z.p;
  {[t]
    if[0=count value t; :()];
    subs[t]@\:(`upd;t;value t);
    fileHandle@(`upd;t;value t);
    numMsgs+:1;
    .[t;();0#];
  } each
  enlist`trade;
  tpflush,:0.001
  *.z.p-tm;
 }
```

Rows per upd	Rows per sec	Timer freq	Pub on timer	Write on timer	tpupd	tpflush	TP CPU	rdbupd	RDB CPU
1	10,000	0	N	N	13	0	31	3	12
1	10,000	100	Y	N	13	36	22	258	0.1
1	10,000	100	Y	Y	3	169	9	273	0.1

Table 4 – Microseconds taken and percent CPU by publish frequency

Where

- Timer freq = frequency the timer in the tickerplant is run in milliseconds
- `tpupd` = median time in microseconds to run `upd` in the tickerplant
- `tpflush` = median time in microseconds to run the timer (`.z.ts`) in the tickerplant

In Table 4 above we can see that when publishing on the timer, the tickerplant `upd` function still takes roughly the same time as in zero latency mode, but we are only publishing data 10 times a second which reduces the overall load: the TP CPU usage has decreased from 31% to 22%. The RDB CPU usage decreases from 12% to 0.1% as it is only doing 10 bulk updates per second instead of 10,000 single updates persecond. Only writing to disk 10 times a second reduces the load on the tickerplant further. The improvements will be greater the more updates the tickerplant receives per second.

8.3.4 Number of subscribers

In this test, we will examine how the number of processes that are subscribing to a tickerplant's data can affect the throughput of the tickerplant. We will run multiple subscribers/RDBs and see the effect on the tickerplant publish time and RDB receive time. We will collect the following metrics:

- Last `rdbrecv` = median time in microseconds for the last RDB in the tickerplant subscription list (`subs` dictionary) to receive the message from the tickerplant
- First `rdbrecv` = median time in microseconds for the last RDB in the tickerplant subscription list (`subs` dictionary) to receive the message from the tickerplant
- Num subs = number of subscribers

Rows per upd	Upds per timer	Timer freq	Rows per sec	Num subs	tppub	First rdbrecv	Last rdbrecv	TP CPU
1	1	1	1,000	1	3	85	85	3
1	1	1	1,000	3	4	172	178	4
1	1	1	1,000	5	6	148	296	6
1	1	1	1,000	10	10	265	343	10
10	1	1	10,000	1	3	88	88	3
10	1	1	10,000	3	5	175	181	5
10	1	1	10,000	5	6	155	318	7
10	1	1	10,000	10	11	224	540	12
100	1	1	100,000	1	21	97	97	6
100	1	1	100,000	3	58	177	324	10
100	1	1	100,000	5	95	257	330	15
100	1	1	100,000	10	185	449	682	30

Table 5 – Microseconds taken by number of subscribers

We can see that increasing the number of subscribers increases the tickerplant publish time, first RDB receive time and last RDB receive time. The first RDB receive time increases because the data is written to each internal message queue and the queues are not flushed until the tickerplant publish function returns.
If there are multiple subscribers to a tickerplant it might be worth considering a chained tickerplant to reduce the number of subscribers. Only the chained tickerplant and the subscribers which need the data as quickly as possible would subscribe to the main tickerplant. Then other subscribers would subscribe to the chained tickerplant to get the data.

8.4 CONCLUSION

This whitepaperexamined some key factors which can influence the performance and throughput of a kdb+ tickerplant. We established amethodology to profilethe performanceof akdb+ tickerplant and used it to focus on four key areas which can affect the tickerplant's throughput:

1. **Number of rows per update**

   Feeds should read as many messages off the socket as possible and send bulk updates to the tickerplant if possible. Bulk updates will greatly increase the maximum throughput achievable,
   e.g. we found processing abulk update of 10 messages took only slightly longerthen processing an update of 1 message, and hence the CPU usage was nearly 10 times lower (for the same number of messages per second).

2. **Size of each update in bytes**

   Reducing the size of the data can improve throughput as writing to disk and sending the data will be faster. The improvement is more noticeable on bulk updates.

3. **Publish frequency**

   Buffering the messages in the tickerplant and publishing on a timer improves throughput. The CPU usage of the RDB also decreases as the data is being batched up into fewer updates per second. However, it is not suitable if the subscribers need the data immediately. Writing the messages to disk on a timer improves throughput further but more data could be lost if the tickerplant dies. The improvement is more noticeable as the number of updates per second increases.

4. **Number of subscribers**

   Adding more subscribers increases the load on the tickerplant. To reduce this, consider using a chained tickerplant. Even existing subscribers will be affected if more subscribers are added since the internal message queues for each subscriber are all written to before the queues are flushed.

It should be noted that these are not the only factors that contribute to a tickerplant's performance. A complete analysis would also examine the effects that network latency and bandwidth, disk writespeed and TCP/IP tuning have on tickerplant performance.

The results shown in this paperare not representativeof kdb+ systems as a whole. Results foreach individualkdb+ system willvary due to various hardware and software considerations. However, the code and methodology used in this paper could serve as astarting point to any developers wishing to profile and optimizetheirown systems.

All tests were run using kdb+ version 3.1 (2014.02.08)

it's about time

Technical Whitepaper

9 Columnar Database and Query Optimization

Author:

Ciarán Gorman is a Financial Engineer who has designed and developed data management systems across a wide range of asset classes for top tier investment banks.

CONTENTS

9.1 INTRODUCTION

The purpose of this whitepaper is to give an overview of some of the methods that are available to a kdb+ developer when trying to optimize the performance of a kdb+ database when queried.

kdb+ has a well deserved reputation as a high performance database, appropriate for capturing, storing and analyzing massive amounts of data. Developers using kdb+ can use some of the techniques outlined in this whitepaper to optimize the performance of their installations. These techniques encompass variations in the structure of queries, how data is stored, and approaches to aggregation.

Adjustments to column attributes are made using the dbmaint.q library, which was written by First Derivatives consultants and is available from the Kx wiki (http://code.kx.com/wsvn/code/contrib/fdsupport/database%20maintenance/dbmaint.q).

Where appropriate, OS disk cache has been flushed using Simon Garland's io.q script, which is available at the Kx wiki (http://code.kx.com/wsvn/code/contrib/simon/io/io.q).

Tests performed using kdb+ version 2.8 (2012.05.29)

9.2 OVERVIEW OF TEST DATA

The data used for the majority of the examples within this whitepaper is simulated NYSE TaQ data, with a minimal schema and stored in a date partitioned db. The data consists of the constituents of the S&P 500 index over a 1-month period, which has in the region of 10,000 trades and 50,000 quotes per security per day. Initially, no attributes are applied, although the data is sorted by sym, and within sym on time. Where an example requires a larger dataset, we will combine multiple partitions or create a new example dataset.

```
q)meta trade
c    | t f a
-----| -----
date | d
time | t
sym  | s
price| f
size | i
stop | b
cond | c
ex | c

q)meta quote
c    | t f a
-----| -----
date | d
time | t
sym  | s
bid  | f
ask  | f
bsize| i
asize| i
mode | c
ex   | c

q)(select tradecount:count i by date from trade)lj(select
quotecount:count i by date from quote)
date      | tradecount quotecount
----------| ----------------------
2010.12.01| 4940021    24713334
2010.12.02| 4940300    24714460
2010.12.03| 4940482    24708283
...
2010.12.30| 4982917    24933374
2010.12.31| 4994626    25000981
```

9.3 QUERY STRUCTURE

Before considering optimizations that can be achieved through amendments to data structures, etc, it is important to approach query construction correctly. This is important in all types of databases, but especially when dealing with on-disk data where the database is partitioned, as poorly structured queries can cause the entire database to be scanned to find the data needed to satisfy the query.

While kdb+ generally evaluates left-of-right, the constraints on `select` statements are applied from left to right following the `where` statement. It is important to make sure that the first constraint in a request against a partitioned table is against the partition column (typically this will be date).

For partitioned and non-partitioned tables, one should aim to reduce the amount of data that needs to be scanned/mapped as early as possible through efficient ordering of constraints.

9.3.1 Query Structure Example

The purpose of this example is to examine the performance of queries with differently-ordered constraints as the database grows. In order to vary the size of the database, we will use `.Q.view`. To measure performance, we will use `\ts` to measure the execution time in ms and space used in bytes.

Note that while the non-optimal requests complete successfully in our relatively small test database, these queries run much slower on a larger database. These queries should be executed with caution.

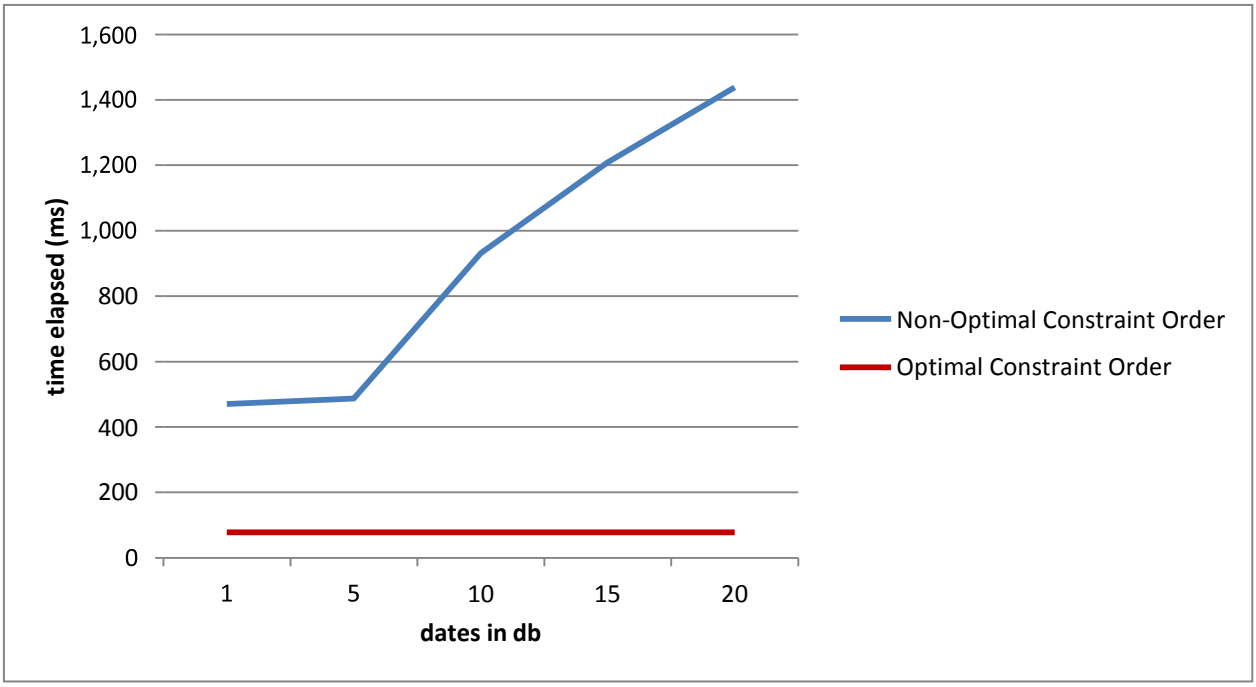
The following queries will be performed, with OS disk cache flushed between requests by restarting the process and using the flush functionality in io.q.

```
//set the date to be requested
q)d:first date

//sym before date (non-optimal order)
q)select from trade where sym=`IBM, date=d

//date before sym (optimal order)
q)select from trade where date=d, sym=`IBM
```

	sym before date		date before sym	
dates in db	time (ms)	size (b)	time (ms)	size (b)
1	470	75,499,920	78	75,499,984
5	487	75,878,400	78	75,499,984
10	931	75,880,624	78	75,499,984
15	1,209	75,882,912	78	75,499,984
20	1,438	75,885,072	78	75,499,984

Table 1 - Results from queries as database grows

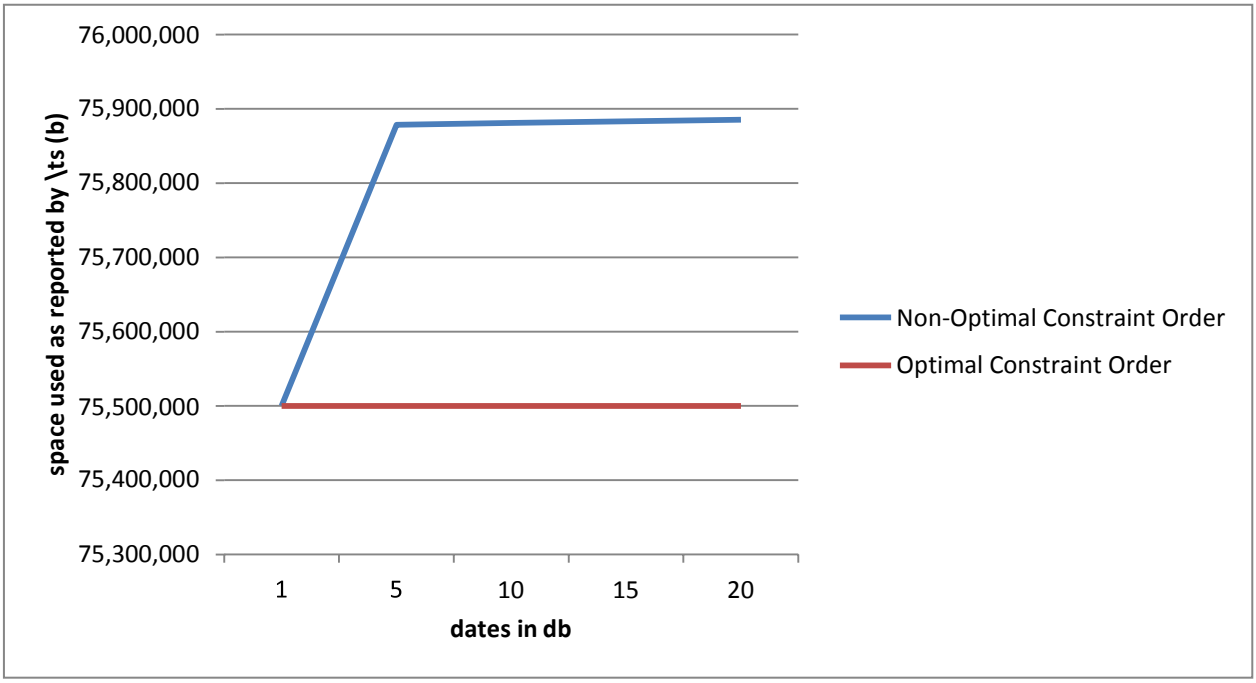
Figure 1: Effect of constraint order on execution times



The results show that with non-optimal constraint order, the elapsed time for queries increases with the size of the db, while the request times for optimal constraint order is constant. The reason for the increase is that kdb+ has to inspect a much larger range of data in order to find the calculation components.

One result of interest is the performance for the non-optimal request in a db containing 1 date; we are only looking at 1 date, therefore one might expect that the performance would be similar. In order to explain the difference here we have to recall that date is a virtual column in the database – inferred by kdb+ by inspecting the folder names in the partitioned db. When the partition column is a constraint, but not the first constraint, kdb+ creates this data for comparison. What we are observing is the cost of this promotion and comparison.

Figure 2: Effect of constraint order on workspace used



The scale of the chart above has been adjusted to show that there is a slight difference in space used as reported by `\ts` for each request. This is because the virtual column is promoted to an in-memory vector of date values for comparison if it is used as a constraint other than the first constraint.

It should be noted that even for optimal constraint order, this is a high amount of space to use, given the size of the result set. We will address this when we look at attributes later.

9.4 PRECALCULATION

For any system, it is advisable to pay particular attention to the queries that are commonly performed and analyse them for optimisations that can be made by precalculating components of the overall calculation. For example, if the db is regularly servicing requests for minute - interval aggregated data, then it probably makes sense to precalculate and store this data. Similarly, daily aggregations can be precalculated and stored for retrieval.

9.4.1 Precalculation As Query Grows

The aim of this example is to show the impact on request times of precalculation and storage of intermediate/final calculation results. We will make baseline requests against our example date-partitioned `trade` table. For the precalculated data, we will create a new table (`ohlc`) with daily aggregated data, and save this as a date-partitioned table in the same database. Between each test, kdb+ will be restarted and the os cache will be flushed.

```
//make dictionary containing groups of distinct syms drawn from our
sym universe
q)syms:n!{(neg x)?sym}each n: 1 10 100 500

//make call for open prices for ranges of security drawn from the
dictionary

q)select open:first price by sym from trade where date=first date, sym
in syms 1
...
q)select open:first price by sym from trade where date=first date, sym
in syms 500
```

Now we have a baseline for values drawn from the quote table, save a table with daily aggregates for each security

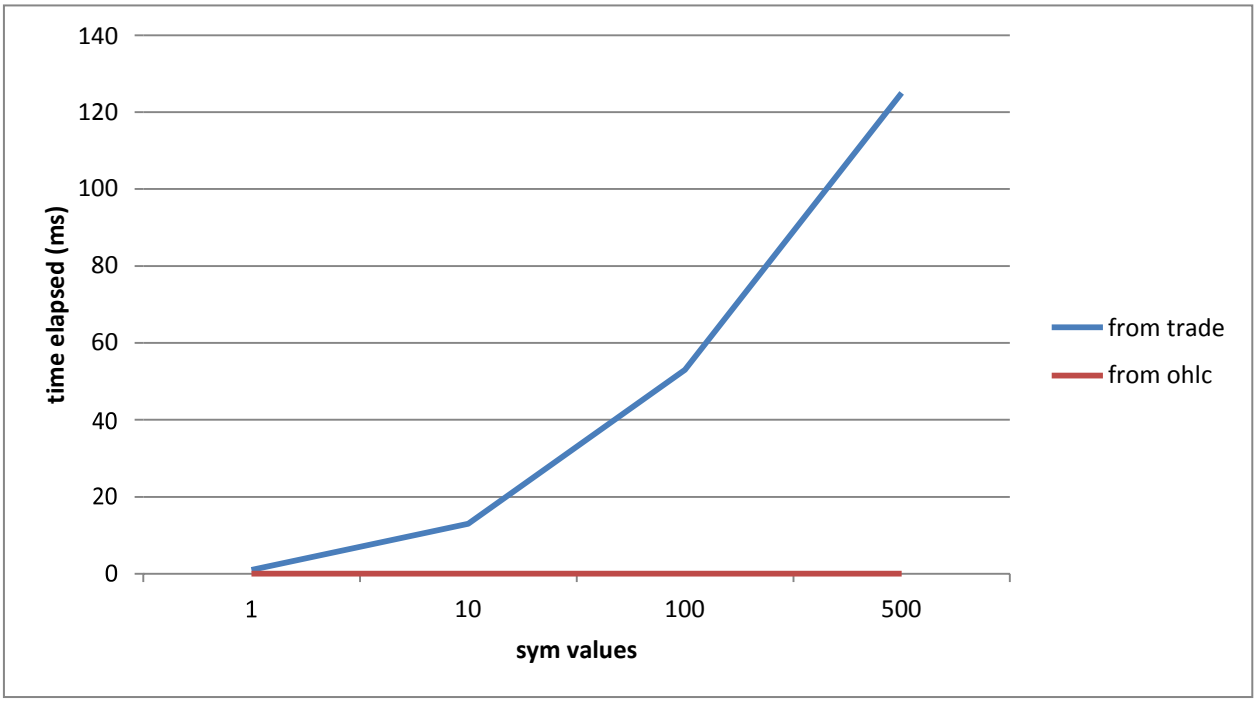
```
//create ohlc table with a number of different day-level aggregations,
and save to disk using dpft function.

//perform across each date in the db
q){ohlc::0!select open:first price, high:max price, low:min price,
close: last price, vwap: size wavg price by sym from trade where date=
x;.Q.dpft[`:.;x;`sym;`ohlc];}each date

//This will create one row per security/date pair.
q)select count i by date from ohlc
date      | x
----------| ---
2010.12.01| 500
2010.12.02| 500
...
2010.12.30| 500
2010.12.31| 500
```

```
//reload db
q)\l .

//repeat the same requests, but target our new aggregation table.
q)select open by sym from ohlc where date=first date, sym in syms 1
...
q)select open by sym from ohlc where date=first date, sym in syms 500
```

	from trade		from ohlc	
sym values	time (ms)	size (b)	time (ms)	size (b)
1	1	265,056	0	2,896
10	13	2,100,224	0	3,504
100	53	16,781,680	0	12,400
500	125	134,228,736	0	48,752

Table 2 - Results from queries as more securities are

requested Figure 3: Effect of using precalculated data on execution times



We can see that for a relatively low expenditure on storage, there are significant performance gains by pre-calculating commonly requested data or components of common calculations.

9.5 TAKE ADVANTAGE OF THE BUILT-IN MAP-REDUCE CAPABILITIES OF kdb+ PARTITIONED DATABASES

For certain aggregation functions, kdb+ has built-in map-reduce capabilities. When a query crosses partitions - for example a multi-day volume weighted average price (VWAP), kdb+ identifies whether the aggregation can be map-reduced. If so, it will perform the map operation across each of the partitions. Some aggregations that support map-reduce are `avg, count, max, min, sum, wavg`.

Map-reduce implementation is transparent to the user. For requests that do not involve aggregation, the map component is simply a retrieval of the required rows, and the reduce combines to form the result table.

9.5.1 Map-reduce As Table Grows

In this example, we will look at the impact of increasing the amount of data being aggregated. We will perform the same query over widening time periods.

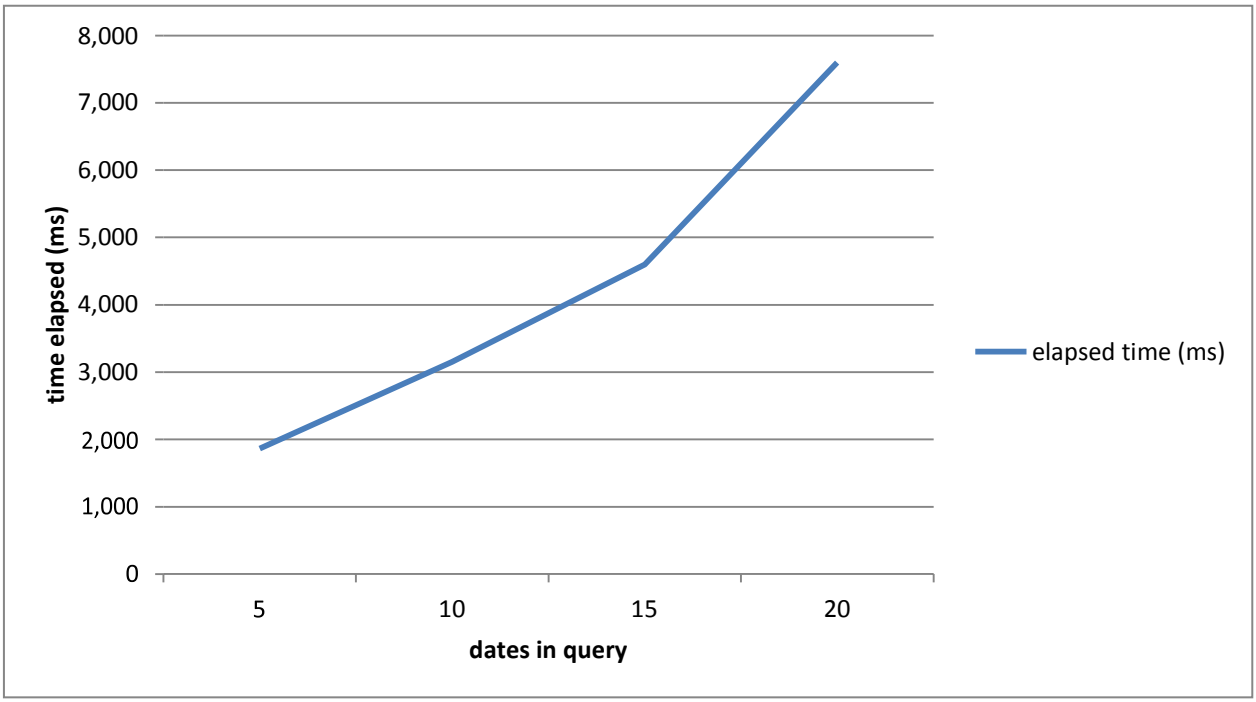
```
//perform the same query over widening date windows
//restart kdb+ and flush os cache between each request

q)select size wavg price by sym from trade where date in 5#date
...
q)select size wavg price by sym from trade where date in 20#date
```

dates in query	time (ms)
5	1,863
10	3,152
15	4,596
20	7,597

Table 3 - Results from queries using map-reduce

When we look at the performance of the requests, we can see that due to map-reduce, the time for the calculation is scaling near-linearly as the number of dates increases. If there were no map-reduce implementation for this aggregation, the cost of mapping in all data then calculating over this dataset would lead to performance worse than linear scaling.

Figure 4: Map-reduce execution times as db grows



It is possible to take advantage of this behavior with kdb+'s slaves, where the map calculation components are distributed to slaves in parallel, which can lead to very fast performance. The use of slaves is outside the scope of this paper.

9.6 USE OF ATTRIBUTES

Attributes can be applied to a vector, table, or dictionary to inform kdb+ that the data has certain properties that allow different treatments to be applied when dealing with that data.

9.6.1 Parted Attribute: `p#

The parted index is typically used to provide fast access to sorted data in splayed on-disk tables, where the most commonly filtered and/or grouped field (often the instrument) has a sort applied prior to saving. It indicates that a vector is organized into contiguous (but not necessarily sorted) chunks.

Applying `p# allows kdb+ to identify the unique values within a vector quickly, and where in the vector the segments begin The result of this is an improvement in the time required to perform a request, and also a reduction in the amount of data required to perform some calculations. The performance improvement will be most noticeable where the values in the vector are sufficiently repetitive. There is a memory overhead to set `p# attribute on a vector, and kdb+ will verify that the data is actually parted as described.

9.6.2 Parted Attribute As Table Grows

If we apply the `p# attribute to the sym columns in the on-disk database we have created for testing and retry the example from the Query Structure example in section 3.1, one can see the benefits of the reduced work that kdb+ has to carry out to find the data it is looking for. We will then be able to compare the results and examine the impact from applying attributes.

In this test, we want to examine the performance of queries applied against data partitioned by date and parted on sym, with the `p# attribute applied to the sym column. The database will be varied in size using `.Q.view`, and between each test we will restart kdb+ and flush the os cache.

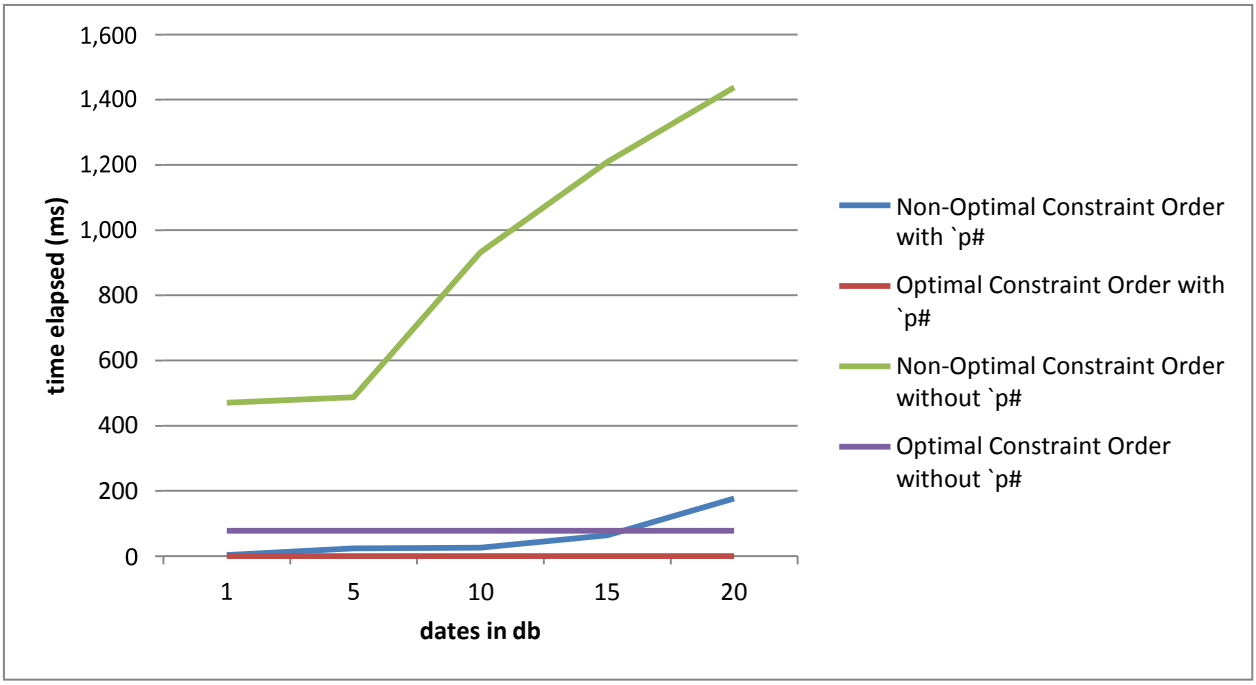
```
//add `p attribute using setattrcol from dbmaint.q
//NOTE: setattrcol does not adjust the order of the data. Our data is
already segmented by sym.
q)setattrcol[`:.;`trade;`sym;`p]
q)\l .

//set the date to be requested
q)d:first date

//sym before date (non-optimal order)
q)select from trade where sym=`IBM, date=d

//date before sym (optimal order)
q)select from trade where date=d, sym=`IBM
```

	no attribute(from 3.1)				`p# set			
	sym before date		date before sym		sym before date		date before sym	
dates in db	time (ms)	size (b)	time (ms)	size (b)	time (ms)	size (b)	time (ms)	size (b)
1	470	75,499,920	78	75,499,984	3	445,024	0	445,088
5	487	75,878,400	78	75,499,984	24	889,936	0	445,088
10	931	75,880,624	78	75,499,984	26	892,816	0	445,088
15	1,209	75,882,912	78	75,499,984	64	896,336	0	445,088
20	1,438	75,885,072	78	75,499,984	177	898,576	0	445,088

Table 4 - Results from queries on data with `p# attribute

As we can see in the table above, when we compare optimal to non-optimal query construction there is still a significant increase in timings and space required when a non-optimal constraint order is used. However, in general the time and space required is much lower than the corresponding examples without `p# applied.

Figure 5: Effect of `p# on execution times

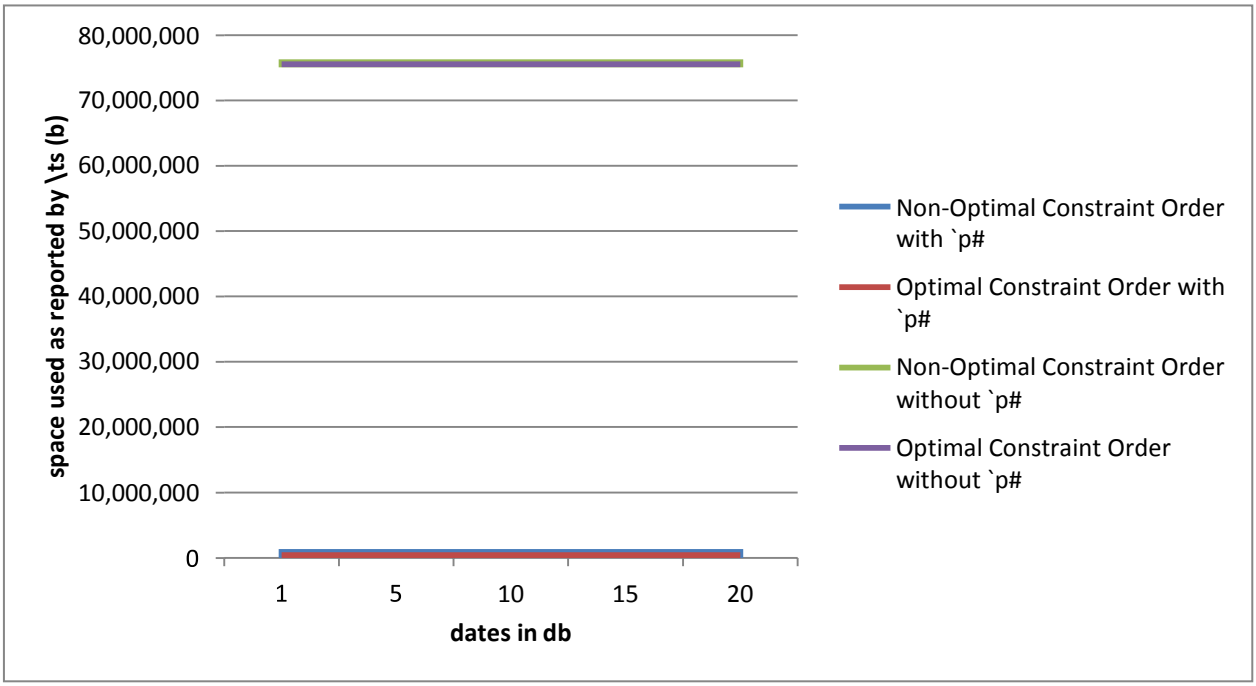


When we look at the results of the `p# and non-optimal constraint order examples, we can see the benefits of both applying `p# and using an optimal constraint order in the query. Initially, both optimal and non-optimal requests against the db with `p# set are faster than their counterparts without attributes. As the database grows, we can see that the optimally ordered request without `p# starts to outperform the non-optimal request with `p# set. The reason for this is that requests with optimal constraint order only have to search within the data for one date. While `p# has optimized the search

within each date for the non-optimal request, the data it has to search through is growing as the db grows, and before long the work being done surpasses the work being done in the non `p# database.

As the db grows it becomes clear that the best-performing version is with `p# applied and optimal constraint order.

Figure 6: Effect of `p# on workspace used



The queries where `p# is set also require significantly less space for calculation than those without `p#. This is because kdb+ is able to move directly to the location in the sym vector that is of interest, rather than scanning sym to find values that match the constraint.

9.6.3 Sorted Attribute: `s#

The sorted attribute indicates that the vector is sorted in ascending order. It is typically applied to temporal columns, where the data is naturally in ascending order as it arrives in real time. When kdb+ encounters a vector with `s# attribute applied, it will use a binary search instead of the usual linear search. Operations such as `asof` and `within` are much faster using the binary search. Requests for `min` and `max` can be fulfilled by inspecting the start and end of the vector, and other comparison operations are optimized.

Setting `s# attribute on a vector has no overhead in terms of memory required, and kdb+ will verify that the data is sorted ascending before applying the attribute.

9.6.3.1 Sorted Attribute As Table Grows

Let's look at the impact on query performance of applying `s# as the dataset grows. We will be applying the `s# attribute to the time column by sorting the data on time as we select it from the on-

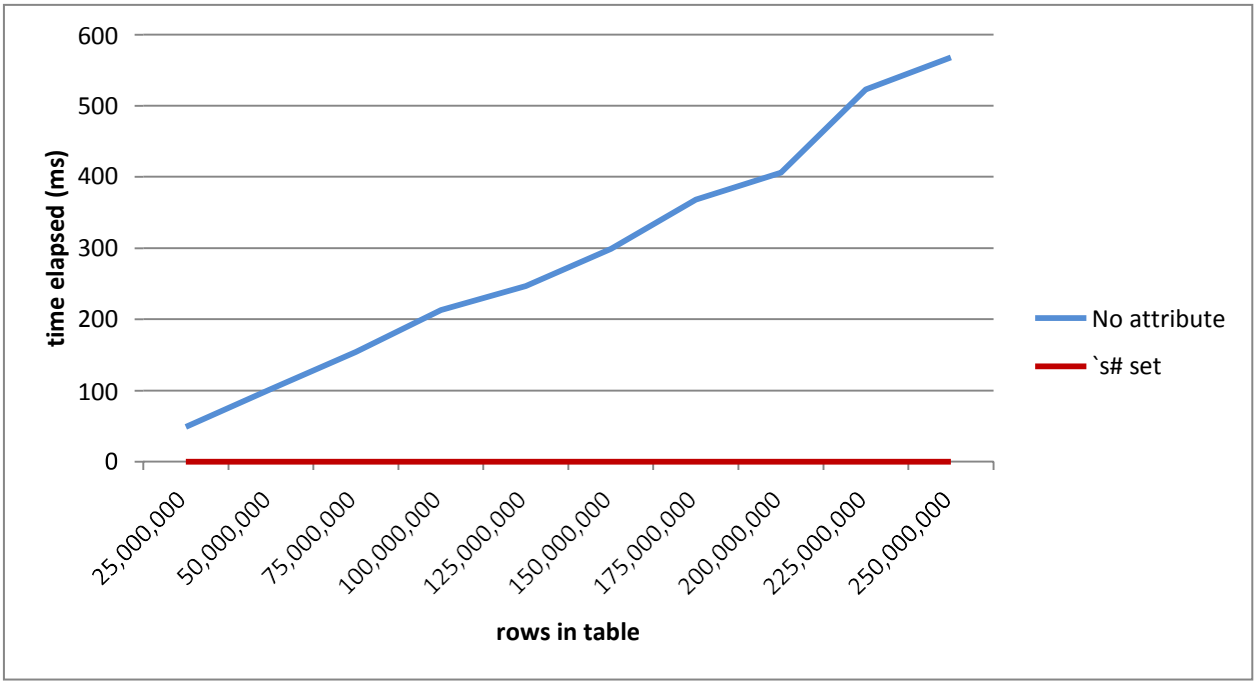
disc data. As we are not interested in the contents of the calculation results, we can combine data from multiple dates in the database and sort on time to simulate growing realtime tables. Between each request, we will restart kdb+.

```
//create time we will use for test requests
q)t:first 09:30:00.000+1?07:00:00.000

//create temporary table using one or more days of saved data – vary from 1 through 10 on each iteration
q)rtquote:{`time xasc select from quote where date in x#date}1
...
q)rtquote:{`time xasc select from quote where date in x#date}10

//make requests against rtquote for IBM data. Repeat as the rtquote table grows
q)select from rtquote where time=t, sym=`IBM

//repeat process with `s removed from time vector
q)rtquote:update `#time from{`time xasc select from quote where date in x#date}1
...
q)rtquote:update `#time from{`time xasc select from quote where date in x#date}10
```

	no attribute		`s# set	
rows in table	time (ms)	size (b)	time (ms)	size (b)
25,000,000	49	33,554,944	0	1,248
50,000,000	102	67,109,392	0	1,248
75,000,000	154	134,218,288	0	1,248
100,000,000	213	134,218,288	0	1,248
125,000,000	247	134,218,288	0	1,248
150,000,000	299	268,436,016	0	1,248
175,000,000	368	268,436,016	0	1,248
200,000,000	406	268,436,016	0	1,248
225,000,000	523	268,436,016	0	1,248
250,000,000	568	268,436,016	0	1,248

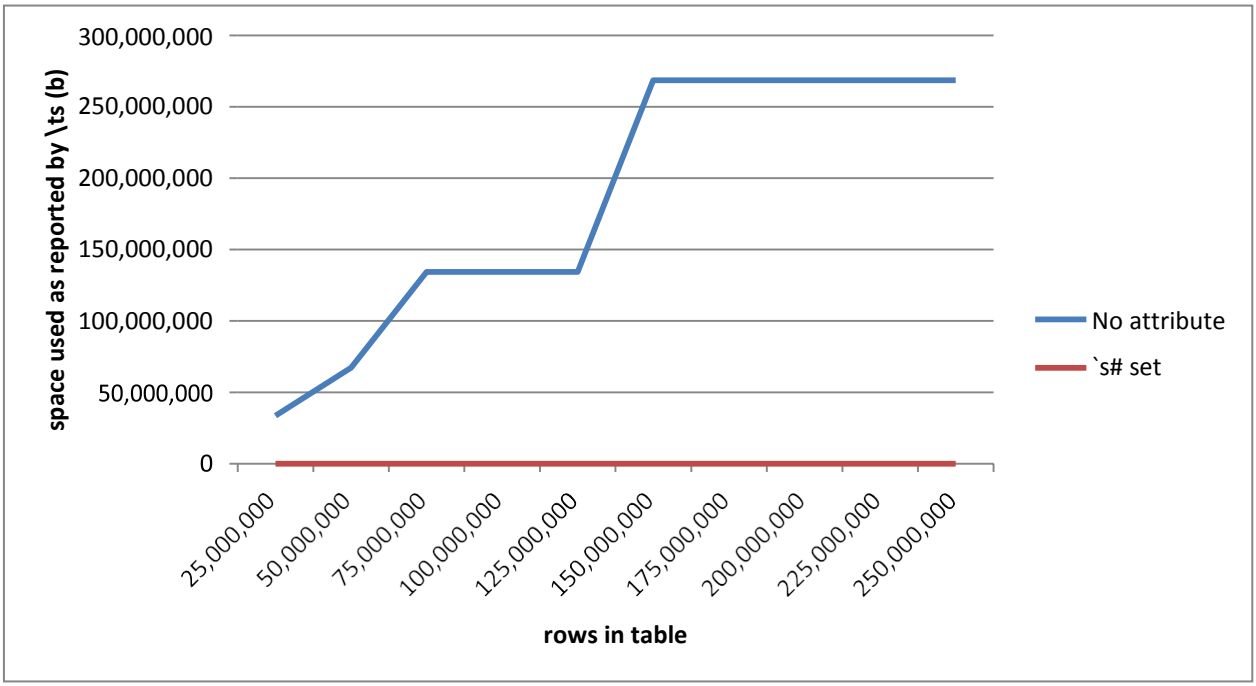
Table 5 - Results from queries on data with `s# attribute

Figure 7: Effect of `s# on execution times



We can see in the chart above the benefits from setting `s# on the time column. While the request times without the attribute grow linearly with the increase in data, the times for the requests against the data with the attribute set are uniform – in this case returning instantly.

Figure 8: Effect of `s# on workspace used



We observe similar results here for the amount of space used for the request. kdb+ is able to perform the calculation with a much smaller subset of the data. We can see that the curve representing space used for requests without an attribute set has a series of steps. These steps reflect the fact that kdb+'s buddy memory algorithm allocates according to powers of 2.

9.6.4 Unique Attribute: `u#

The unique attribute is a hash map used to speed up searches on primary key columns, or keys of simple dictionary maps, where the list of uniques is large and/or lookups are very frequent. There is a memory overhead for applying `u# to a vector, and kdb+ will verify that the data is unique before applying the attribute.

9.6.4.1 Unique Attribute As Table Grows

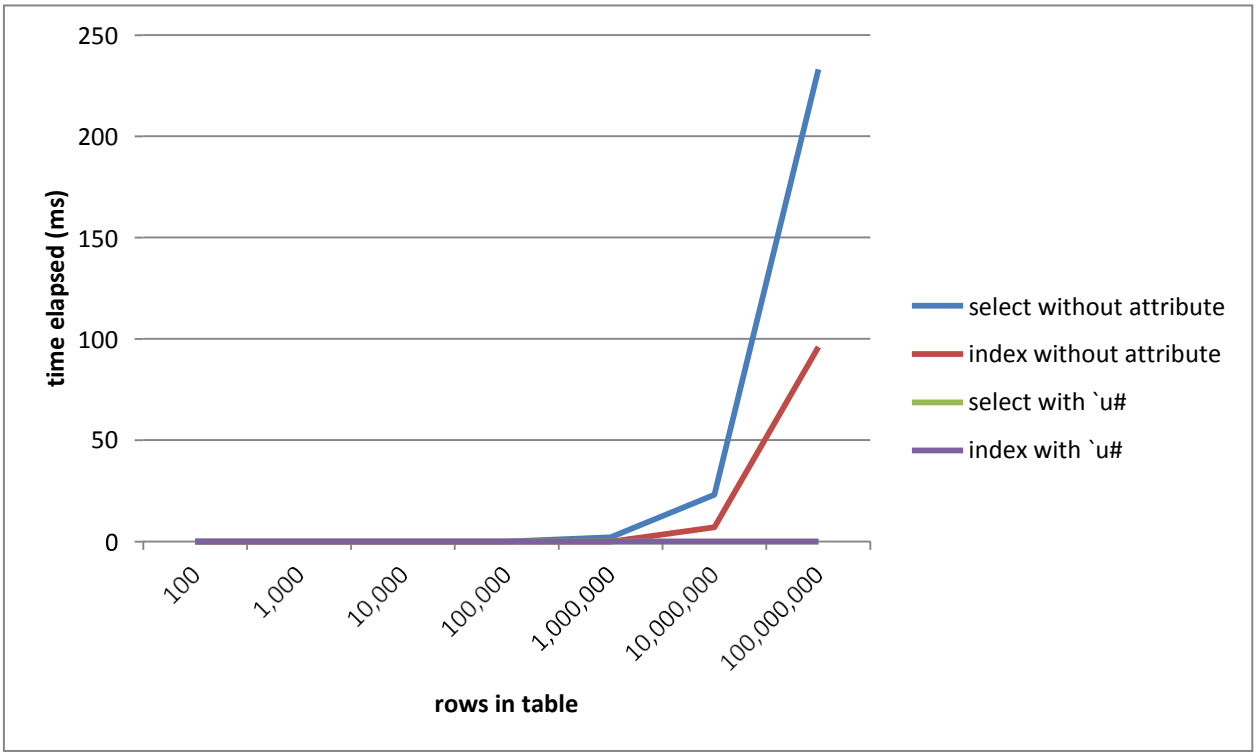
This example will examine the performance of the `u# attribute as the dataset being queried grows. The example data we've used for the previous examples does not contain values with a high enough degree of uniqueness to be used here. We will create some new data for this example using the q `rand` function. The table that we will use for this example will be similar to a last value cache, although to observe the behaviour, we will probably grow it beyond a reasonable size for tables of this form.

```
//function to create example last value cache keyed table with x rows
//also creates global s to retrieve query value later
q)mktbl:{([sym:s::(neg x)?`7];lprice:0.01*x?10000)}

//make example table, vary through the values on each iteration
q)tbl:mktbl 100
...
q)tbl:mktbl 10000000

//create test value
q)req:first 1?s

//select value
q)select lprice from tbl where sym=req

//directly index to retrieve value and create table
q)enlist tbl[req]

//repeat with `u# applied to sym column
q)tbl:update `u#sym from mktbl 100
...
q)tbl:update `u#sym from mktbl 10000000
```

	no attribute				`u# set			
	select		index		select		index	
rows in table	time (ms)	size (b)	time (ms)	size (b)	time (ms)	size (b)	time (ms)	size (b)
100	0	1,040	0	528	0	1,040	0	528
1,000	0	1,536	0	528	0	1,040	0	528
10,000	0	16,896	0	528	0	1,040	0	528
100,000	0	131,584	0	528	0	1,040	0	528
1,000,000	2	1,049,088	0	528	0	1,040	0	528
10,000,000	23	16,777,728	7	528	0	1,040	0	528
100,000,000	233	134,218,240	96	528	0	1,040	0	528

Table 6 - Results from queries on data with `u# attribute as table grows

Figure 9: Effect of `u# on execution times as table grows



The chart above shows that as the table size crosses 1m data points, the time taken for requests against non-attributed data starts to grow. Requests against attributed data remain uniform in response time, in this case returning instantly.

In addition, it can be observed that directly indexing into the keyed table with no attributes applied is faster and uses less data than performing a `select` against it.

9.6.4.2 Unique Attribute as Query Grows

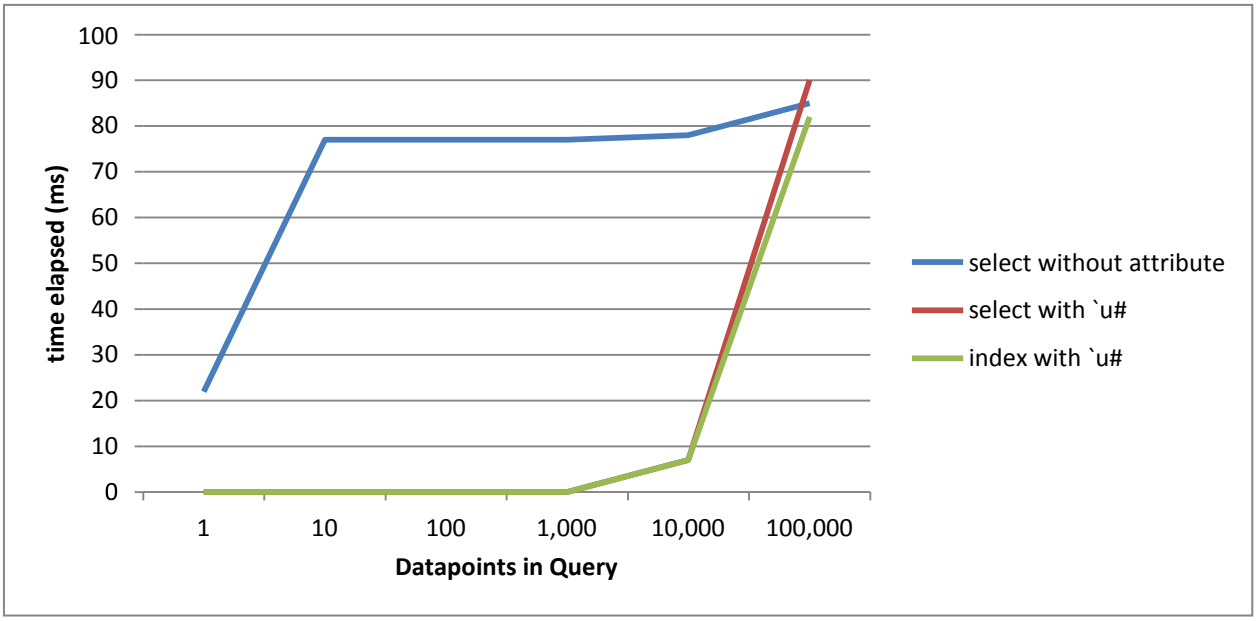
For this example, we'll create a static last value cache table and grow the number of sym values being requested.

```
//function to create example last value cache keyed table with x rows
//create global s to retrieve query value later
q)mktbl:{([sym:s::(neg x)?`7];lprice:0.01*x?10000)}

//create fixed size example table
q)tbl:mktbl 10000000

//make dictionary containing groups of distinct syms drawn from our
sym universe
q)syms:as!{(neg x)?s}each as:1 10 100 1000 10000 100000

//select value
q)select lprice from tbl where sym in syms 1
...
q)select lprice from tbl where sym in syms 100000

//directly index to retrieve value and create table
q)tbl each syms[1]
...
q)tbl each syms[100000]

//repeat with `u# applied to sym
q)tbl:mktbl 10000000
q)update `u#sym from `tbl
```

	no attribute				`u# set			
	select		index		select		index	
sym values	time (ms)	size (b)	time (ms)	size (b)	time (ms)	size (b)	time (ms)	size (b)
1	22	16,777,776	0	608	0	1,344	0	608
10	77	83,886,624	60	1,392	0	1,344	0	1,392
100	77	83,886,624	462	9,872	0	6,288	0	9,872
1,000	77	83,886,624	7,454	88,976	0	53,008	0	88,976
10,000	78	83,886,624	48,822	1,033,616	7	648,208	7	1,033,616
100,000	85	83,886,624	484,486	9,546,128	90	5,821,968	82	9,546,128

Table 7 - Results from queries on data with `u# as query grows

Figure 10: Comparison of query times with `u# attribute as query size grows



NB: In order to observe the lower points on the curve, the 'index without attribute' curve is not charted.

As we can see, selects against vectors with `u# applied significantly outperform those without an attribute applied. As the queries grow, we can see that there can be limitations to the performance gains with large queries. For the index syntax, the reason for this is that our example code is written in such a way that we are operating in a scalar fashion across the requested sym vector.

This example is provided as commentary, not as an example of recommended design. As mentioned previously, `u# is typically applied to aggregation tables or dictionary keys, so it is unlikely to be required to service requests of the size we reach in this example. In a scenario where there is a large query domain, it may be faster to break up into smaller queries and combine the results.

9.6.5 Grouped Attribute: `g#

The grouped attribute is a conventional database 'index' of all instances of a particular value. The ability of kdb+ to maintain this in real time as data is inserted allows fast searching, grouping and filtering on important fields.

Applying the grouped attribute to a column causes the regular search algorithm to be substituted for a hash based search. This allows developers to identify the unique values within a vector quickly, and to quickly retrieve the values required. There is a significant memory/disk cost for applying `g# to a vector.

9.6.5.1 Grouped Attribute As Table Grows

This example is concerned with observing the performance of setting `g# on the sym column of a realtime quote table, which is typically ordered by time. We will observe the performance of locating matched through an unordered sym vector.

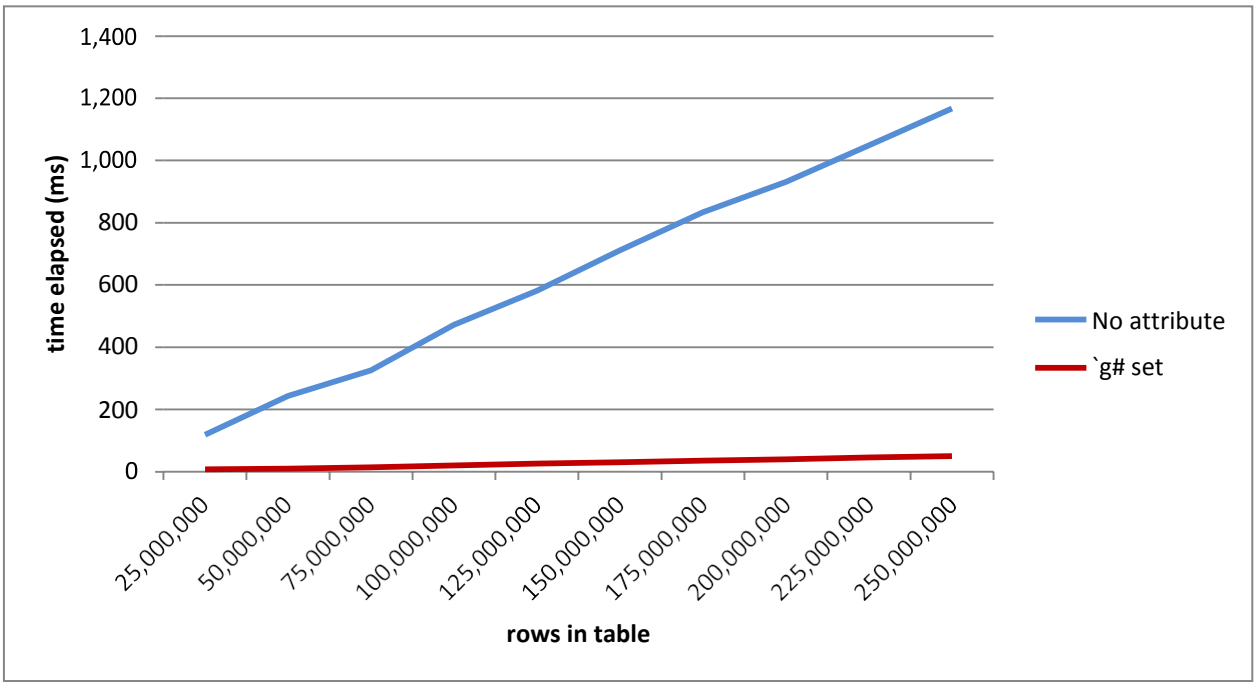
As we are not interested in the contents of the calculation results, we can combine data from multiple dates in the database and sort on time to simulate growing realtime tables. We will not set any attribute on the time column, and will restart kdb+ between queries.

```
//create rt quote table with `g#sym. Vary through 1-10 dates
q)rtquote:update `g#sym, `#time from `time xasc select from quote
where date in 1#date
...
q)rtquote:update `g#sym, `#time from `time xasc select from quote
where date in 10#date

//select all IBM data from throughout the table
q)select from rtquote where sym=`IBM

//repeat the process for data without `g#sym
q)rtquote:update `#time from `time xasc select from quote where date
in 1#date
...
q)rtquote:update `#time from `time xasc select from quote where date
in 10#date
```

	no attribute		`g# set	
rows in table	time (ms)	size (b)	time (ms)	size (b)
25,000,000	119	301,990,304	8	2,228,848
50,000,000	243	603,980,192	10	4,457,072
75,000,000	326	1,207,959,968	14	8,913,520
100,000,000	472	1,207,959,968	20	8,913,520
125,000,000	582	1,207,959,968	26	8,913,520
150,000,000	711	2,415,919,520	30	17,826,416
175,000,000	834	2,415,919,520	36	17,826,416
200,000,000	931	2,415,919,520	40	17,826,416
225,000,000	1,049	2,415,919,520	46	17,826,416
250,000,000	1,167	2,415,919,520	50	17,826,416

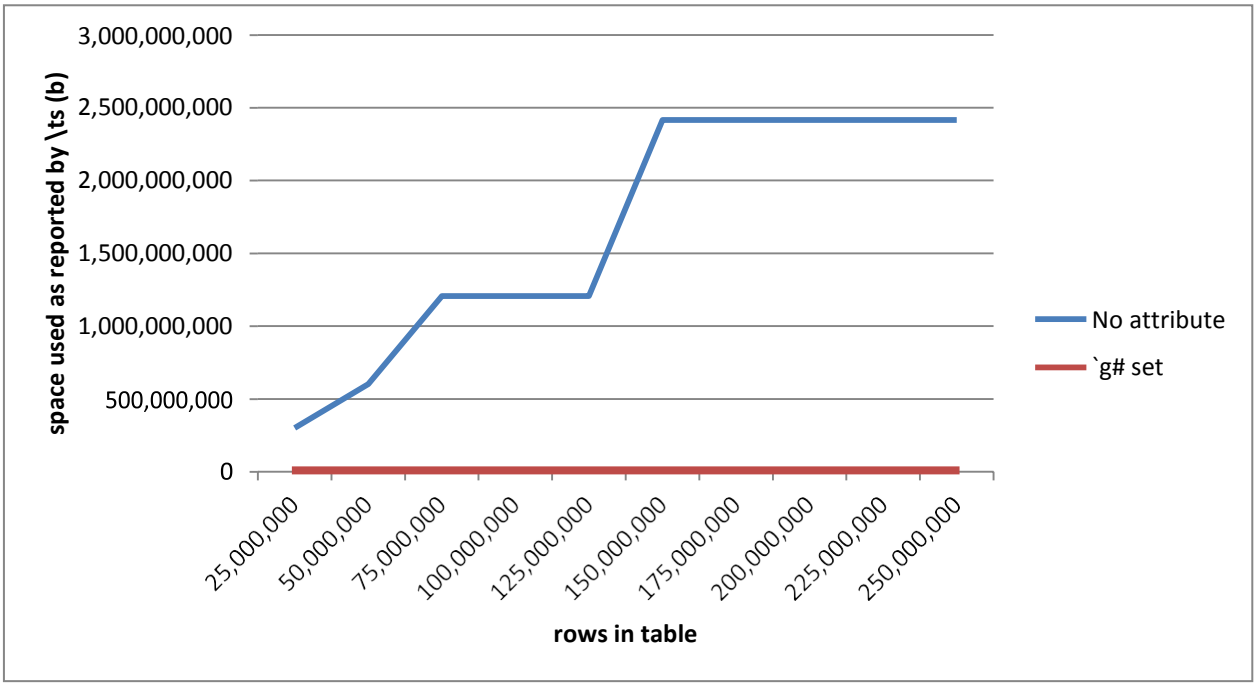
Table 8 – Results from queries on data with `g# attribute as table grows

Figure 11: Effect of `g# on execution times as query grows



We can see from this example that even when sym values are distributed across the table, having `g# applied to the sym vector allows for a significant speedup.

Figure 12: Effect of `g# on workspace used as table grows



The chart showing space used shows a similar pattern to the timing curve. The slight increase in space used is because the result set is growing as the size of the table increases.

9.6.5.2 Grouped Attribute to Retrieve Daily Last

As discussed, setting `g# causes the regular search algorithm to be replaced with a hash-based search. We can make use of this hash directly using the group function with the column that has `g# applied.

To illustrate this, consider the following methods for calculating the last price for each symbol for a day of simulated real-time trade data. We will restart between examples:

```
//create simulated realtime trade table sorted on time
q)rttrade:update `#time from `time xasc select from trade
where date=first date
q)\ts select last price by sym from
rttrade 24 33561280j

//create simulated realtime trade table sorted on time
q)rttrade:update `g#sym, `#time from `time xasc select from
trade where date=first date
q)\ts select last price by sym from
rttrade 15 33559232j

//create simulated realtime trade table sorted on time

q)rttrade:update `g#sym, `#time from `time xasc select from
trade where date=first date
//use group on the sym column to get the indices for each sym and find
//last value in each vector.  Use these index values to directly index
//into the price vector, then create a table with the results
q)\ts {1!([]sym:key x;price::value x)} rttrade[`price] last each
group[rttrade`sym]
0 15072j
```

As we can see in the results above, using the same form of query with the attribute applied results in a significant speedup. However, using an alternative approach in which we retrieve the individual indices for each security, then find the last, results in a much faster calculation, and usage of much less data.

9.6.5.3 Grouped Attribute as Query Grows

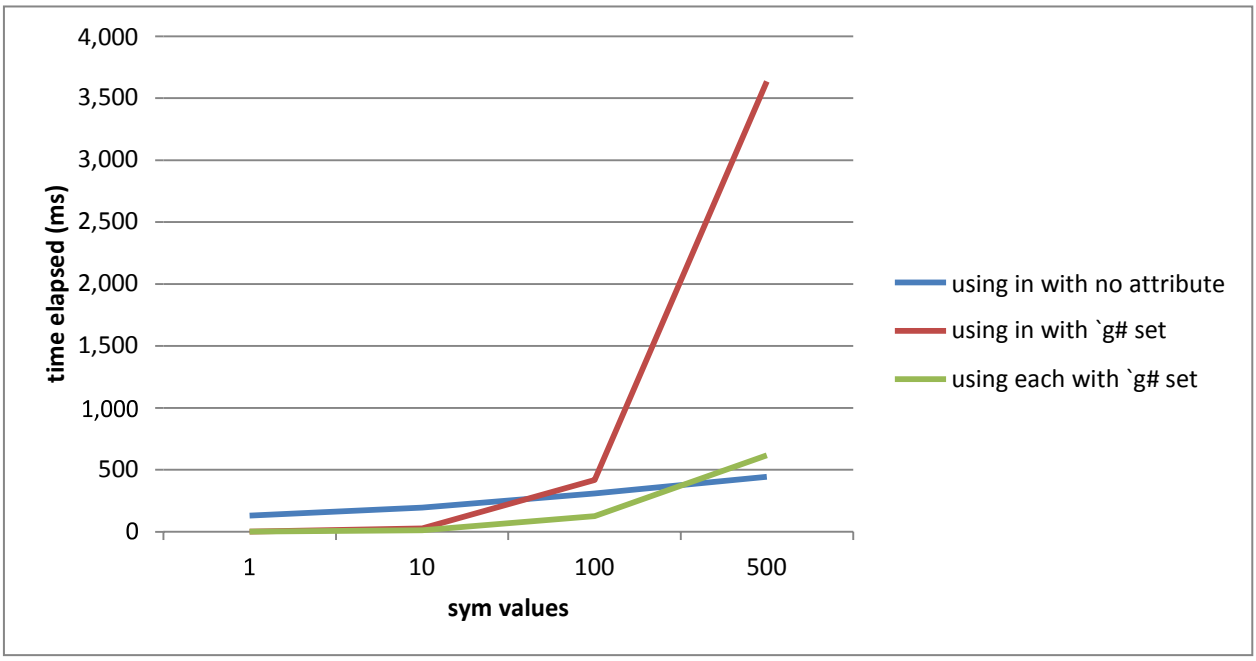
This example will examine the behaviour of `g# as the requested universe grows. We will use more than one query format, and observe whether the format of the request impacts performance. The data used for the example is quote data drawn from our example database, and sorted on time to simulate realtime data. kdb+ will be restarted between tests.

```
//make dictionary containing groups of distinct syms drawn from our
sym universe
q)syms:n!{(neg x)?sym}each n:1 10 100 500

//create rt quote table with `g#sym
q)rtquote:update `g#sym, `#time from `time xasc select from quote
where date=first date

//comparing the performance of different query forms as we increase
the number of securities requested
q)select first bid by sym from rtquote where sym in syms 1
...
q)select first bid by sym from rtquote where sym in syms 500


q)raze{select first bid by sym from rtquote where sym=x} each syms 1
...
q)raze{select first bid by sym from rtquote where sym=x} each syms 500
```

	No attribute				`g# set			
	using in		using = and each		using in		using = and each	
sym values	time (ms)	size (b)	time (ms)	size (b)	time (ms)	size (b)	time (ms)	size (b)
1	132	301,990,624	133	301,991,104	1	787,408	1	787,904
10	197	402,653,920	1,319	301,993,792	29	12,583,696	13	790,592
100	309	402,653,920	13,204	302,020,608	417	201,327,824	125	817,408
500	444	536,879,984	27,484	33,965,504	3,634	536,879,984	616	935,680

Table 9 - Results from queries on data with `g# attribute as query grows

Figure 13: Comparison of query times with `g# attribute as query size grows



NB: omitted curve for 'using = and `each`' without attributes in order to examine more closely the faster-returning examples.

As we can see from the chart above, there are points at which it has been more performant to use `=` for comparison and loop over the universe of securities being requested, then join the result sets using `raze`. This is because `select` preserves the order of records in the table, so has to coalesce the indices from the `` `g# `` hash records for each security into a single ascending index when using `in` and a list, but this step is not necessary when using a function over a list of securities.

Note: If an aggregation on sym is being performed, using a function and operating over the list of values will result in the same data set, but if an aggregation is not being performed (e.g. `select from rtquote where sym = x`), then the result sets will differ – the result set for the function format will have data grouped into contiguous sym groups.

9.7 CONCLUSION

This paper has described a number of different techniques that are available to kdb+ developers when trying to optimise the performance of a kdb+ database when queried. We have looked at query structure, precalculation, map-reduce, and various effects from applying attributes to data. With each of these topics, we have attempted to provide an indication of the reasons for variations in the performance of kdb+, and identify some limits and edge cases associated with these techniques.

It is important to point out that the use of these techniques should be married to an appropriate overall system design – taking into account the data structures they are being applied to, the type of access required, and any outside factors (hardware constraints, etc.). They should not be considered magic bullets that can be used to rescue an inefficient design – doing this will likely lead to larger problems in the longer term. It is recommended that developers experiment with the techniques outlined above - and additional performance techniques that we have not explored in this paper - to find the approach that best fits the use case being handled.

Tests performed using kdb+ version 2.8 (2012.05.29)

it's about time

Technical Whitepaper

10 Multi-Partitioned kdb+ Databases: An Equity Options Case Study

Author:

James Hanna, has helped design and develop kdb+ implementations and proof of concepts for more than 40 customers. Based in New York, James is a Technical Architect for Kx, a high performance data management, event processing and trading platform.

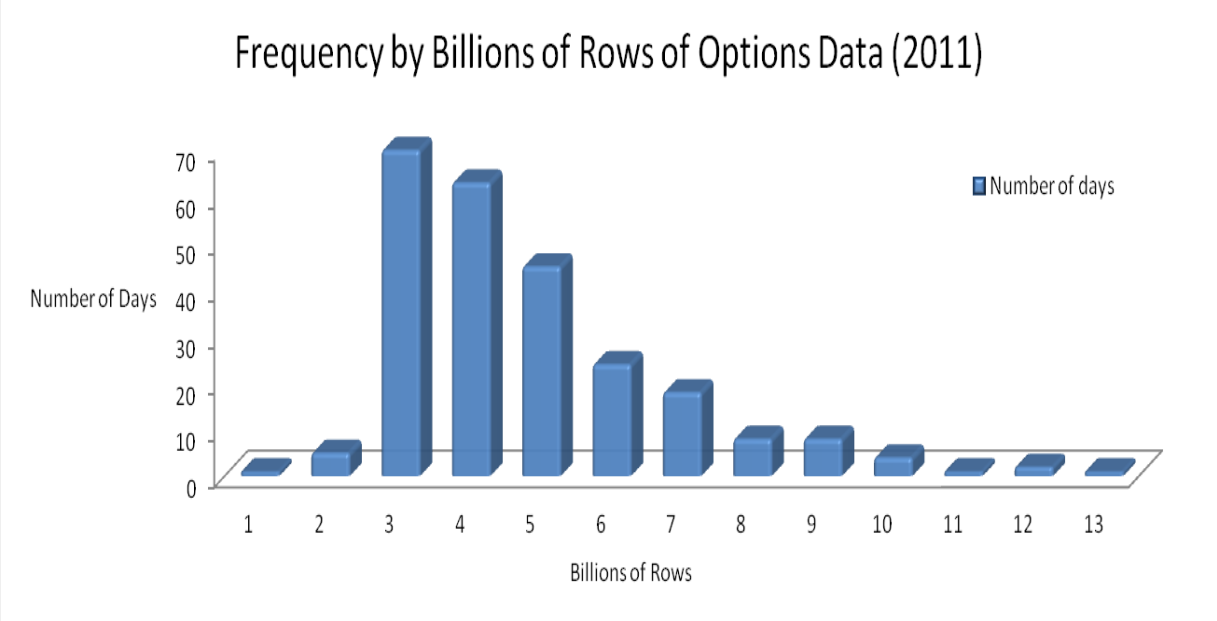
CONTENTS

10.1 INTRODUCTION

Kdb+ is well suited to managing massive datasets and offering an unrivalled performance advantage when it comes to processing and analyzing data. This is a case study highlighting some of the key points we have found with regard to the storage and maintenance of financial equity options data in kdb+. We also provide some examples of possible ways to design and query these large databases efficiently.

10.1.1 Overview of the Dataset

The Equity options data universe is one of the largest financial datasets generated, and more often than not the most challenging dataset for trading firms to manage and extract value from in a timely manner. Hundreds of gigabytes of trade and quote records are published daily from the equity options feeds, with recent daily row count volumes for Q1 of 2012 having an average and maximum count close to 4 billion and 6 billion rows respectively. These numbers represent a relief of sorts from the peaks of 2011 where we saw a maximum daily row count of over 13 billion as highlighted in the chart below.



The dataset we use for our examples in this paper includes level 1 trade and quote data for financial options contracts, the associated reference data and the corresponding equity level 1 trades and quotes for the same time period. All data has been sourced from tickdata.com.

The sample dataset has a maximum of approximately 200 million rows for a single date and includes data for 10 underlying securities. The full universe that our clients load is typically much larger, covering thousands of underlying securities. For some of these securities, for example AAPL, the ratio of the number of option quotes to underlying quotes can be in excess of 60:1 on some dates.

It is thus inevitable that when storing equity options data for a large universe of underlying securities, the number of rows per partition will regularly exceed 2 billion. When kdb+ 3.0 was released in 2011 it removed the limit to the number of rows that can be stored in a single partition. This gives us two options for storing massive tables, either storing each day of data in a single partition, or storing each day of data in multiple partitions.

In this paper we cover the use of a multi partitioned database, as whilst kdb+ 3.0 allows the very straight forward option of having a single partition for each date, there are still potential advantages to the approach of storing data across multiple partitions for a single date.

Firstly, when saving data, multiple partitions can be written concurrently, potentially reducing the time required to load data from flat files or persist it from memory. Secondly, when running queries against the database, data from multiple partitions can be read in parallel using slave threads. In the same way as queries on a database with a single partition per date can read data for multiple days in parallel, now data for the same date can be read in parallel. A third advantage related to database maintenance is that since the size of the individual column data files is reduced, the memory required to apply a sort to the data on disk will be reduced.

Kdb+ provides a simple method to store the data in multiple partitions for each date by using the par.txt file. When attempting to read data from a database like this a large number of queries will behave exactly as they would in a database with only one partition per date. However there are some cases in which it's necessary to rework things a little, most notably in the case of asof joins. This will be covered below.

10.2 OPTIONS SCHEMA

Sorting and indexing the options data is straightforward. If we have the data sorted by underlying security, option contract and finally timestamp we can apply the partitioned attribute to both underlying security and option contract, allowing us to filter quickly on either column.

More interesting is the way in which we choose to store the underlying security market data so that we can link it to the options data efficiently. The simplest way of accomplishing this would be to store the prevailing value of each underlying alongside the option trades and quote data. These extra columns would either be provided in the source data (as was the case in our sample dataset) or could be pre- calculated with an asof join and then stored.

The advantage of this method is that we do not need to store any additional underlying market data and there is no overhead when performing lookups to find the prevailing underlying quote or last trade for an option quote or trade. However there are two obvious downsides to using this approach.

The first is that it does not offer any flexibility with regard to how the market data for an underlying security maps to the corresponding option data, e.g. if we wanted to do a window join to look at multiple underlying quotes surrounding each option quote. The second is that a significant amount of extra storage space will be required when the data is de-normalized in this way compared to storing the underlying market data in separate tables and doing joins on demand. Even with a frugal schema for the underlying data, this might add 40-50 bytes of storage for each record (depending on whether condition code and exchange fields can be stored as single characters or symbols are required for example). Given that there will be billions of option quotes per day, this can add hundreds of gigabytes to the daily storage requirements. It is worth noting that this may not be as large a problem as it first appears given the possibilities for data compression.

A second possibility is to store underlying market data as completely separate tables and do any joins we require between the two datasets on a purely ad hoc basis. This option offers the lightest storage requirements and also gives us full flexibility in how we do the joins. It does, however, come with the cost of extra processing time when searching for the underlying market data related to the option data at query time. Examples of how this is done can be found in the queries section below.

A third option, a combination of the first two, would be to save the option and underlying data as separate tables, but to compute row indices in the underlying market data tables and store them in the option tables as link columns. This requires less space (a single integer column per link to each underlying table) than storing full underlying trade/quote information along with each option table row, avoids having to find the correct underlying trade/quote at query time and also gives flexibility by having all the underlying data available for ad hoc joins.

Using the third option requires us to ensure that the underlying and option data for the same securities always reside in the same partition. We can achieve this as part of the load process outlined below.

10.2.1 Loading and Saving Data

Here we assume some familiarity with loading large data files by splitting the file and loading in chunks using .Q.fs and .Q.fsn. For more details please refer to this section on code.kx.com *http://code.kx.com/wiki/Cookbook#kdb.2B_dealing_with_big_data.*

10.2.2 Striping data over multiple partitions per date

Firstly we demonstrate how to save a chunk of loaded and parsed data into a database with multiple partitions per date. In our case we will split the data alphabetically by underlying symbol into groups as follows:

ABC, DEF, GHI, JKL, MNO, PQR, STU and VWXYZ

We should have a par.txt file containing paths to directories for each of these symbol groups. The contents of our par.txt file are thus as follows:

```
/data/0
/data/1
/data/2
/data/3
/data/4
/data/5
/data/6
/data/7
```

Before demonstrating how we can stripe the data over these directories, it is worth noting that if future data volumes increase and we wish to partition the data into a greater number of stripes, we can do this by adding new directories to par.txt without the need to go back and repartition old data. We would only need to create empty tables for each existing date for each partitioned table in our db.

With regard to saving a chunk of data, there are many ways to do this but we provide an example below where we read in a list of available partitions and create some helper functions to allow us to easily partition and save the data. In the code below the directory is assumed to be a global variable (DIR) giving the path to our par.txt file in q format (symbol with a leading colon).

```
// A dictionary mapping alphabetical group to the directory
// in the database we wish to save to
dirs:`ABC`DEF`GHI`JKL`MNO`PQR`STU`VWXYZ!hsym each`$read0 ` sv
DIR,`par.txt

// A function which will return a list of partitions
// to which each of a list of symbols should be
saved. getpart:.Q.fu {[symlist]
  key[dirs]0 3 6 9 12 15 18 21 bin .Q.A?first each string symlist,()}

saveonepart:{[dt;tablename;data;part2save]
 (` sv dirs[part2save],(`$string dt),tablename,`)set
 .Q.en[DIR]
 delete part from select from data where part=part2save}
```

We could use the following snippet to save some quote data stored in a variable CHUNK inside a function called by .Q.fs or .Q.fsn. DATE is assumed to be a global variable here representing the date for which we are loading data. It is elementary to modify this to include the possibility of data for more than one date in a particular file load. This is excluded here for the sake of simplicity.

```
{
..
 CHUNK:update part:getpart underlyingSym from CHUNK;
 saveonepart[DATE;`QUOTE;CHUNK]each distinct exec part from
 CHUNK;
..
}
```

By implementing the previously defined getpart function to generate the partition to save down to based on either the underlyingSym column for options data or the sym column for underlying market data, we can be sure that related options and underlying data will be stored in the same partitions.

Once the data has been loaded and saved we will often need to apply an attribute to one or more of the columns of the data. This is a fairly easy step as we just need to apply the attribute to a table spread over multiple directories instead of one. In our example of option quotes we would like to apply the partitioned attribute (`p#) to both the sym and underlyingSym columns. We can do this as follows:

```
// function applies p# attribute to sym and underlyingSym columns
// of the quote table for the specified date and
directory addphashes:{[dt;dir]
 {[dt;dir;f]@[` sv dir,(`$string dt),`QUOTE;f;`p#]}[dt;dir]each
  `sym`underlyingSym}
```

This may be called after loading all the data in our script:

```
addphashes[DATE]each value dirs;
```

10.2.3 Adding links to market data

If we wish to store links to the underlying data within the option data, a sensible time to generate and store these links would be just after we have loaded, sorted and added attributes to the data. We can do this as part of a loader script with the following code. Here we just create one link stored in the QUOTE table to the corresponding quote in the EQUOTE (underlying quote) table:

```
dirs: `$read0 ` sv
DIR,`par.txt
addlinks:{[dt;dir]
 dir:` sv dir,`$string dt;
 // compute links as an asof
 join. inds:select ind: x from
 aj[`sym`timestamp;
  select sym:underlyingSym,timestamp from
  dir`QUOTE; select sym,timestamp,i from
  dir`EQUOTE];
 // save the links
 (` sv dir,`QUOTE`underlying)set `EQUOTE!exec ind from inds;
 // update the metadata of the QUOTE table
 u set distinct get[u:` sv dir,`QUOTE`.d],`underlying}
```

Again we should use this for each partition for the date we have just

loaded. `addlinks[DATE]each value dirs;`

10.3 EXAMPLE QUERIES

In all of the examples below variables in caps are used instead of specific values.

10.3.1 Raw options quote data retrieval with underlying quote

In the simplest case where we have the prevailing underlying quote stored alongside the option quote as columns lastbidPrice and lastaskPrice our query is:

```
select sym, timestamp, bidPrice, askPrice, lastBidPrice, lastAskPrice
from QUOTE where date=DATE, sym=SYM, time within (STARTTIME;ENDTIME)
```

For the dataset where we have links to the prevailing underlying quote stored we can use:

```
select sym, timestamp, bidPrice, askPrice, underlying.bid,
 underlying.ask from QUOTE where date=DATE, sym=SYM, time within
 (STARTTIME;ENDTIME)
```

In the case where we have the options data and underlying data stored in separate tables, we could usually use an asof join. However, since our database has multiple partitions per date, specifying the right most argument of the asof join in the usual way does not behave in the same manner as in the case of databases with a single partition per date. For example, if we look at a standard query to get the prevailing quote as of each trade we encounter some problems.

```
aj[`sym`time;
  select price from TRADE where date=SOMEDATE, sym in
  SYMLIST; select sym, time, bid, ask from quote where
  date=SOMEDATE]
```

In the case of a database with a single partition per date, this asof join does not read the entire sym, time, bid and ask columns into memory before performing the lookup, rather it searches for the correct rows from a memory map of the quote table. In a database with multiple partitions per date, the following part of the previous query proves problematic:

```
select sym, time, bid, ask from EQUOTE where date=SOMEDATE
```

This now causes all of the data to be mapped and read into memory so that the asof join can be performed. Not only is this undesirable due to the extra I/O required but because multiple memory mapped partitions are being collapsed into one in memory table, it also has the detrimental side effect of removing the partitioned attribute normally found on the sym column since rows with the same symbol could occur in multiple partitions. This is not something that should actually occur in this database but since it is possible, q will defensively remove the partitioned attribute. In addition, because part of the reason we chose a database with multiple partitions for each date was to ensure no single partition was bigger than 2 billion rows, we may even hit a limit error when trying to do asof joins against large tables (this could only occur when using a version of kdb+ older than 3.0).

The result of the above is that not only would a join take longer because more data is being read from the disk but also the join itself will be performed much more slowly. A particular technique to work around this would be to create our own function which will perform joins against each memory mapped partition for a certain date without reading the full table into memory, then aggregate the results from each partition.

In the function below the parameters are specified as follows:

- c is a list of column names upon which to perform the asof join on just as we would have with a regular aj
- t1 is the first table, similarly just as in the case of a regular aj

- t2 is the second table name. This is a symbol giving the name of a partitioned table. To do an asof join on a non-partitioned table a regular aj should be used
- t2d is the date upon which to do the join. We should do one date at a time if there are multiple dates
- t2c is a list of column expressions for the selection from the partitioned table, e.g. (`Ticker;`Timestamp;`Trade;`TradeSize;(log;`TradeSize)). This will usually just be some subset of the column names in the table
- t2cn is a list of column aliases in parted table e.g. `Ticker`Timestamp`price`size`lnprice

```
ajparted:{[c;t1;t2n;t2d;t2c;t2cn]
 if[not all c in
 t2cn;'`missingcols];
 / we want just one row per row in the input, so put a row id as a key
 / and just fill values in the input table with
 / rows that are found for the second table
 / build table a table with the right schema to start
 t1:`rid xkey update rid:`s#i from
 aj[c;t1;?[t2n;enlist(<;`date;first
 date);0b;t2cn!t2c]];
 / do aj's on each partition of the second table with the first table,
 / only return rows that we have values for,
 / then upsert these to the keyed
 t1 delete rid from 0!t1,/
  {[c;t1;t2n;t2d;t2c;t2cn;x]
   inds@:vinds:where not null
   inds:
    (c#M:?[(` sv`$string x,t2d)t2n;();0b;t2cn!t2c])bin (c#value
   t1); 1!(0!t1)[vinds],'M inds}[c;t1;t2n;t2d;t2c;t2cn]
 peach distinct .Q.pd}
```

Returning to our original example query of getting the prevailing underlying quote information for a particular set of option quotes, we can use the following call to the ajparted function:

```
t1: select sym,timestamp,bidPrice,askPrice from QUOTE where
date=SOMEDATE,sym in SYMLIST,timestamp within (STARTTIME:ENDTIME)
```

```
ajparted[`underlyingSym`timestamp;t1;`EQUOTE;SOMEDATE;`sym`timestamp`bid`ask
;
`underlyingSym`timestamp`bid`ask];
```

10.3.2 Snapshot of option chain

This is an example where we have an underlying symbol and want to get a snapshot of the quotes available for all options at a given time.

Firstly, we will need to query the security master table (called mas in our case) for all option data available on the given date for this underlying and then find the last quote available for each option at, or prior to, the given time.

The available option contracts may be queried from a security master table, mas, using the following:

```
optsyms:select sym from mas where date=OURDATE, underlyingSym=OURSYM
```

Now that we have all of the options contracts we require, the question becomes how to query the QUOTE table to get the available quote for each option at the given time. One way to do this would be to write a query to extract the last quote prior to the time in question for each option:

```
select last bid,last ask by sym from QUOTE where date=OURDATE, sym
in optsyms, time<=OURTIME
```

However, this is another place where we would normally use an asof join since it allows us to efficiently search for the last record prior to the time in question rather than scan through all of the records for each symbol to see if the timestamp constraint is fulfilled. We can use the same function from the previous example to do an asof join here. Firstly, we use the cross function to create a table of symbol and time pairs in order to carry out the asof join:

```
rack:optsyms cross ([]timestamp:1#OURTIME)
```

Now we can use ajparted to find the correct rows from the QUOTE table

```
ajparted[`sym`timestamp;rack;
 `QUOTE;OURDATE;`sym`timestamp`bid`ask;`sym`timestamp`bid`ask]
```

In the previous example we were able to avoid an ad hoc asof join provided we had underlying data (or pre calculated links to it) stored alongside the options quote data. In this case however, we will need to use an asof join regardless of how the data is stored.

10.3.3 Building a minutely time series of at-the-money option contracts

In our final example, we demonstrate the creation of a minutely time series based on the idea of a generic option contract. In this case we choose a condition specifying this contract as the next expiring call contract nearest to at-the-money.

Initially, we create a time series of minutely bars for the underlying symbol so we can determine which actual option contract we should use at each point. Given we are working with relatively low volume trade data, we do this with a regular grouping and aggregation query. Note that we could also use ajparted with a rack of symbol and time pairs as in the above example.

```
bars:select last price by time.minute from
 ETRADE where date=ourdate,sym=OURSYM
```

We now need to find the particular contracts that are available on our chosen date and the one that is closest to at-the-money at each point in our series.

We can wrap up the logic for choosing a contract into a function with parameters for date, underlying security, contract type (call or put) and price:

```
closest2atm:{[d;s;t;p]
 / A list of all the nearest expiring contracts on this
 date cands:`strike xasc select sym,strike from mas
   where date=d,underlyingSym=s, typ=t,expir=(min;expir)fby underlyingSym;
 / A list of all strike prices with midpoints between,
 / we can then use bin to find the contract with a strike price to our
 prices searchlist:1_raze{avg[x,y],x}':[cands`strike];
 inds:searchlist bin p;
 / Any odd indices in inds mean price is closer to strike above
 / add one to these and divide everything by 2 to give the indices into
 cands inds[where 1=inds mod 2]+:1;
 inds:inds div 2;
 / return the list of at the money
 symbols cands[`sym]inds}

update sym:closest2atm[OURDATE;OURSYM;”C”;price] from `bars
```

Finally we query the closing bid and ask for each of these bars:

```
ajparted[`sym`timestamp;select
sym,minute,timestamp:`timespan$minute,price from
bars;`QUOTE;OURDATE;`sym`timestamp`bid`ask;`sym`timestamp`bid`ask]
```

10.4 COMPRESSION

The use of compression for all or part of the dataset here is outside the scope of this whitepaper, however, this is undoubtedly an important feature to consider using for options data. The FAQ section on code.kx.com is a good starting point on this topic:
http://code.kx.com/wiki/Cookbook/FileCompression.

10.5 CONCLUSION

Storing and querying options data can present challenges due to the volume of data involved. Whilst the introduction of kdb+ 3.0 lets clients easily handle the ever increasing data volumes by removing the per partition limit of 2 billion rows, there are still several potential benefits to storing the tick data in multiple partitions per date as outlined in Overview of the Dataset. As we have seen however, in choosing this approach it will be necessary to write some queries in a different way to achieve the best performance.

Regardless of whether the data is stored in a single partition per date or in a multi-partitioned manner, the choice of schema will still impact the storage space required for the data and queries against the database. Here we have presented several different schema choices and examples of several simple queries implemented for each schema.

it's about time

Technical Whitepaper

11 Surveillance Technologies to Effectively Monitor Algo and High Frequency Trading

Author:

Sam Stanton-Cook, Ryan Sparks, Dan O'Riordan and Rob Hodgkinson

CONTENTS

11.1 RISING TECHNOLOGIES AND THE ROLE OF SURVEILLANCE

Algorithmic trading has been utilised globally since the implementation of matching engines in modern exchanges. Such technological advances have increased the capacity of markets to process orders and trades by removing human limitations. As a result, the timescale of the market shifted from seconds to milliseconds and the market surveillance moved from the trading pit to computers. The purpose of market surveillance, whether by a regulator or exchange, is to maintain market integrity and protect the participants against unethical behaviour.

Once again, there is a technology race that is surfacing out of the algorithm-dominated market place. Speed is king in the high frequency trading (HFT) game where the participants now operate on microsecond precision. Analytical algorithms that do not require high frequency speeds such as Pairs, Stat Arb and Mean Reversion can process in 200-500 microseconds whilst remaining effective. High frequency systems require latencies as low as 20-50 microseconds to execute speed-orientated strategies. These systems rely upon sophisticated software, hardware and a large investment in exchange located servers to shave nanoseconds off message sending and receiving times. The practice of high frequency trading is observed globally in the capital markets, accounting for 60-70% (Knowledge@Wharton, 2009) of the liquidity in some U.S. stocks while more recently it was predicted to account for around a third of all trades in the Australian markets (ASIC, 2013). High frequency trading in Australia is less prevalent and relatively innocuous compared to other global markets, highlighted in the article published on FinExtra (Finextra, 2014) and detailed in the findings of ASIC's investigation into HFT and dark pools (ASIC, 2013). However the real concern is the lack of sophisticated tools to accurately monitor and profile high frequency and algo patterns, which is the topic of this paper. For regulators to be effective in evolving markets the surveillance software must evolve with changes in market behaviour. In the case of high frequency trading, the system needs to be capable of capturing and processing data at the same speeds.

Kdb+ is a time-series relational database language that combines both a programming language and the underlying database. It is therefore an ideal tool for analysing and identifying high frequency trading, providing an easy way to analyse and compare real time data with large amounts of historical data. The concise and interpreted language enables rapid development, which is integral for working with a dynamic environment such as the HFT world. Furthermore, an important part of any analysis is determining effective parameters for functions – having high speed access to market data and the ability to process it in real time means that back testing for calibration purposes can be achieved quickly and easily. Similarly, once an automated method for identifying HFT behaviour is implemented, all the tools are there for more "manual" verification.

All tests performed using kdb+ version 3.1 (2014.02.08)

11.2 DEMAND FOR SURVEILLANCE

An increase in high frequency related incidences has shaken investor confidence and raised global concerns about market stability and integrity. The 'Flash Crash' in 2010 showcased that high frequency strategies can have rogue tendencies in the right market conditions.

High frequency trading has also been associated with unethical trading practices and in some instances market manipulation. In 2010 Trillium Capital was prosecuted and fined $1 million ($USD) for market manipulation (Comstock, 2010). Since then other high frequency brokers have been seen to systematically manipulate the price of a stock by using techniques such as quote-stuffing and layering. The intention of these techniques is to create a false image of the market depth by placing orders that they never intended to fully trade on. By utilising the low latency of the high frequency trading system the orders can be cancelled before the market has time to respond.

Unrivalled speed allows these traders to detect large orders entering the market on one exchange and react by purchasing or selling the stock on another to capitalise on their knowledge of a participants intention to buy or sell. The practice is commonly referred to as front running and has always had a presence in the marketplace. The premise of front running is identifying a definite buyer (someone who shows intent to buy), buy from a seller and immediately sell to the buyer at an inflated price, all the while carrying no risk. The introduction of smart order routers and improvements to low latency trading systems has enabled high frequency traders to implement this strategy at a new level. Hedge funds and banks have created algorithms to break up their client's larger orders in an attempt to prevent HFTs from front running. Even so, it was estimated that $1.9 billion ($AUD) was lost to high frequency traders acting as intermediaries between two brokers, taking the riskless profits (Sydney Morning Herald, 2014).

To ease the effects of rogue algorithms and to stop crash events from reoccurring, exchanges and regulators have implemented preventative measures like price circuit breakers where the exchange-matching engine has preconfigured limits. When these limits are hit, trading is then suspended for that product. Protecting fundamental investors and day traders from the effect of high frequency trading is not as simple. To restore market confidence, the behaviour of high frequency traders must be understood and seen by the public to be monitored. Identifying and categorizing these traders is the first step. However, it is difficult to separate these traders from the rest of the market. Detecting these traders is achievable by applying a series of simple metrics to historical or real-time data. The identification of these traders allows further investigation of potential unsavoury behaviour.

11.3 DETECTING HIGH FREQUENCY TRADERS

To understand the nature of high frequency trading, a surveillance solution as sophisticated as the trading systems is required to monitor the HFT behaviour in real-time. A kdb+ system can capture and analyse live data to efficiently track these traders using a handful of common metrics. The metrics serve two purposes. Firstly, the goal is to identify potential high frequency traders. The aim is then to determine whether any of the identified traders are negatively impacting the market. For a description of the tables and columns referred to in the following sections refer to Appendix B. All images in the document have been created using Dashboards for Kx, that visualizes the historic and real-time data by interfacing with kdb+.

- *Order to trade ratio (OTR)* – The order to trade ratio metric calculates the total number of order messages divided by the number of trades at a broker, client or account level. It identifies traders that are amending or cancelling orders at a far higher rate than they are trading. High frequency traders are usually identified to have a ratio greater than 15. The heat map below illustrates 200 Australian brokers with ratios ranging from 7.29 to 1434.67. The goal of the metric can be tweaked and focused by examining the max OTR in 10-minute buckets or scanning for bursts in OTR throughout the trading day.

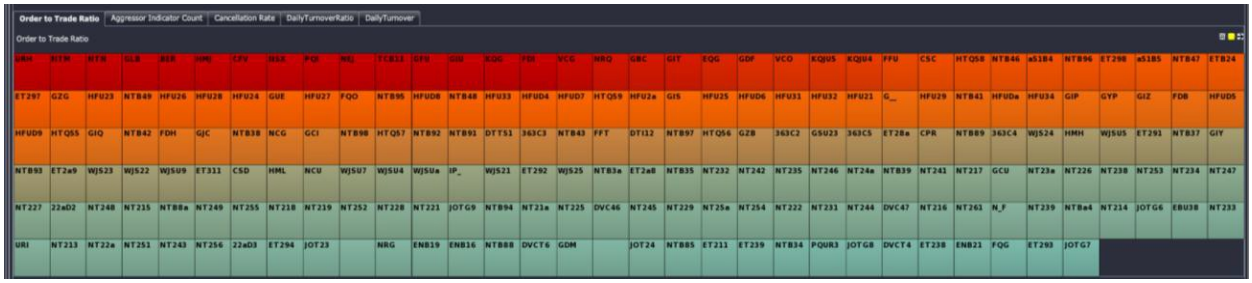


Figure 1 Heat map of the Order to Trade ratio (OTR) for 200 brokers visualised via Dashboards for Kx. Approximate colour scheme: red > 1000 > orange > 15 > blue > 0. All codes have been scrambled.

```
// - Realtime order to trade ratio q code. Calculation based on the last x minutes of
the market

OrderToTradeRatio:{[x]
update OrderToTrade:OrderCount%TradeCount from
      lj[
            select OrderCount:count i by brokerID from dxOrder where
time>.z.P-“u”$x;
                  (select TradeCount:count i by brokerID:buyBrokerID from
dxTrade where time>.z.P-“u”$x) +
                  (select TradeCount:count i by brokerID:sellBrokerID from
dxTrade where time>.z.P-“u”$x)
      ]

// - Select the trade count by broker on the buy and sell side from dxTrade and add
the keyed tables together.
// - Count the number of orders from the dxOrder table and left join (lj) the trade
count table to create a table comprised of brokerID, OrderCount and TradeCount
```

- *Cancellation rates* – This metric is designed to detect a technique known as 'fishing' whereby HF traders rapidly create and cancel orders to test the market within the spread. To calculate the cancellation rate we count the number of orders that each broker created and cancelled under a specified holding time. The example below illustrates the number of orders that each broker held for less than 1 millisecond. While the technique is not necessarily manipulative, it can intentionally coerce other participants and their algorithms to enter the market. Seeing as these orders are unlikely to be filled by fundamental investors they add no benefit to the liquidity or spread. Additionally the high message turnover can be disruptive to the market and its participants. High cancellation rates within the spread can be indicative of quote-stuffing (described in section 4).

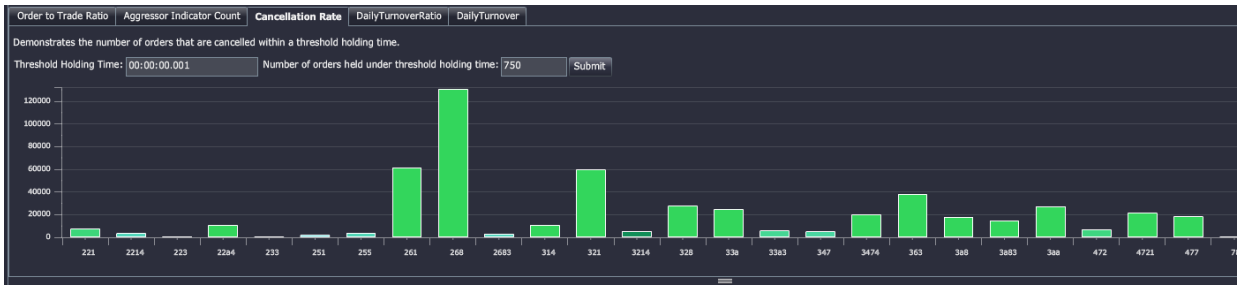


Figure 2 Column chart displaying the number of orders per broker that were held for less than 1 millisecond visualised via Dashboards for Kx. The selection only shows brokers that had at least 750 orders cancelled within the specified holding time. All codes have been scrambled.

```
// - Cancellation rate calculation

select count i by brokerID from dxOrder where orderType in `new`cancelled,
                00:00:00.001 > 1D^ ({x – prev x};transactTime) fby orderID

// - Use fby in the where clause to group transactTime per orderID
// - Apply the function {x-prev x} to each orderID's list of transactTime to calculate
the time between each message
// - Count each message in the table by brokerID where the time between the message
and the previous message (by orderID) is less than 1 millisecond
```

- *Daily Turnover* – Huge traded volumes and short holding times are a major indicator for high frequency activity. HF Traders will avoid holding overnight risk so it is common to close most positions before the end of day. This allows us to look back historically and search for brokers that have closed out all positions in a number of stocks. A position is considered closed out if the volume bought is equal to the volume sold. Counting the number of stocks that are completely closed out within a day will also identify day traders. Therefore the purpose of this metric is to provide a list of possible HF traders that can be further investigated.

  The image below illustrates a range of brokers that bought and sold equal parts of at least 14 stocks in the same day. The size of the bubble represents the number of stocks that were completely closed out and the colour indicates the turnover within the day. This value can be adjusted to look for traders that partially closed out positions. By drilling into the activities of a specific trader (i.e. the highlighted green bubble below) the total volume bought and sold within the day can be examined.

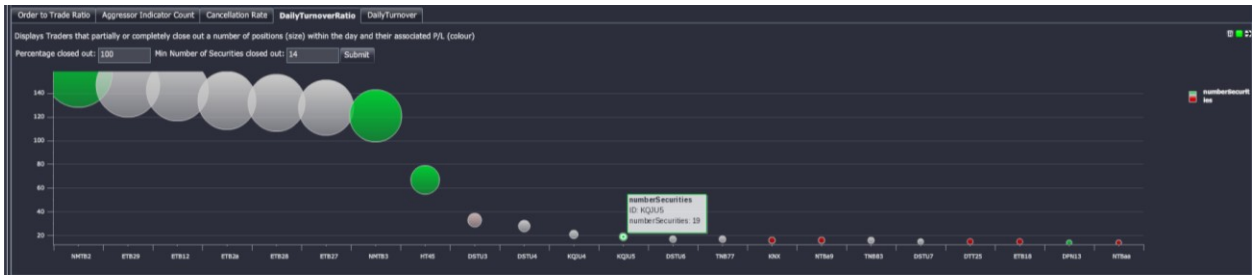


Figure 3 Turnover ratio to identify traders with high close out positions (typically day traders). The bubble size indicates the number of stocks that were entirely closed out within the day while the colour represents the daily Profit and Loss. Image visualised via Dashboards for Kx.

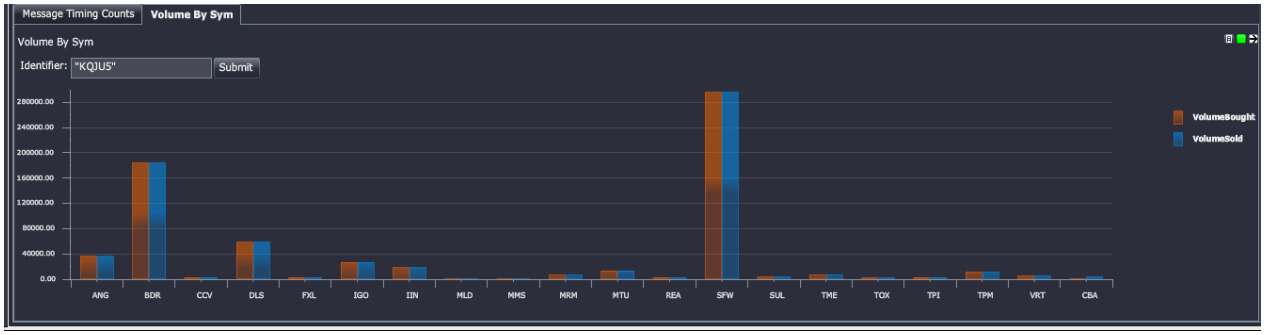
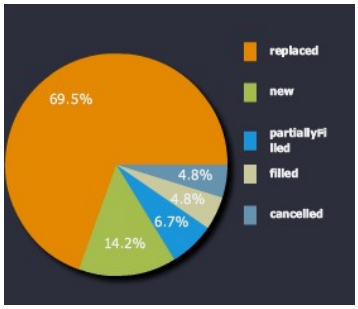


Figure 4 Volume bought and sold for a broker that had a very high turnover ratio visualised via Dashboards for Kx. In this example it is clear to see almost all of the stocks were close outs (equal bought/sold).

- *Message Profiling* – HF Traders can be further categorized by tracking the types of messages and the rate that a person of interest can send them. By calculating the time in between each message from a person of interest a profile can be drawn that details the messaging capabilities of a trader. Traders that demonstrate a left skewed message rate histogram (majority of orders under 20ms) are likely to be high frequency traders. Order distribution of a trader or participant assists in identifying frequent cancellations and 'intent to trade' but perhaps with very few trades actually generated.



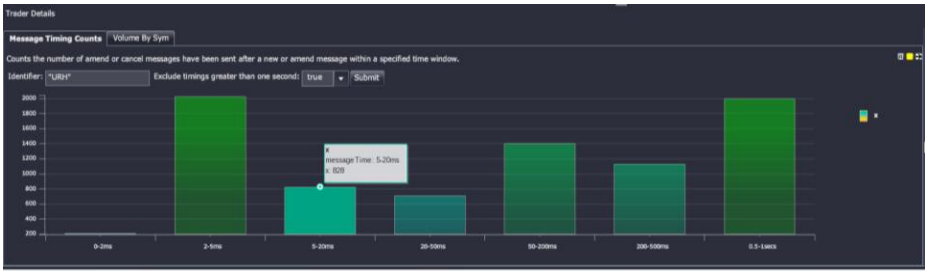


Figure 5 Message profiling. Left chart shows proportion by orderType (New, cancelled, replaced etc.) and right chart shows the message rates as a frequency histogram by message latencies (sub-second). Images visualized via Dashboards for Kx.

```
// - Time profiling code. Bucket sizes and labels
timekey : `$("0";"0-2ms";"2-5ms";"5-20ms";"20-50ms";"50-200ms";"200-
500ms";"0.5-1secs";">1secs");

timebuckets : 0D00:00:00 0D00:00:00.000000001 0D00:00:00.002 0D00:00:00.005
0D00:00:00.02 0D00:00:00.05 0D00:00:00.2 0D00:00:00.5 0D00:00:01;

// - Query to produce the result in the right image of Figure 5
select NumberOfMessages:count i
    by TimeBucket : timekey timebuckets bin transactTime-prev transactTime
from dxOrder where brokerID=`PersonOfInterest, orderStatus in
`new`replaced`cancelled

// - Use the bin function to group the timespans between orders into set buckets.
// - Apply the timekey list to give each bucket a sensible name
```

By focusing on parties of interest regulators can better understand their individual trading techniques and monitor the market more effectively.

11.4 EXAMINING POTENTIALLY HARMFUL STRATEGIES

11.4.1 Quote Stuffing

Quote-stuffing is a technique whereby a trader floods the market with orders and rapidly cancels them. The 'noise' created by the influx of orders can affect other trader's latency and create an arbitrage opportunity for the high frequency trader. If the orders are 'stuffed' within the spread, other traders can perceive a false impression of the market. Quote-stuffing generally occurs in bursts and will last less than 10 seconds on average (Credit Suisse, 2012).

```
// - Function to pull back the quote table on a date (dt) for any stock (s)
// - burstperiod is the length of time we expect the quote-stuffing to occur
// - pricemovements is the minimum number of movements that have to occur within the
burstperiod

quoteFluctuationSym:{[dt;s;pricemovements;burstperiod]

      select from dxQuote where date=dt,sym=s,
      pricemovements < ({[x] sum (<>':) x};bid) fby (burstperiod xbar
transactTime.second)

 };

// - Pull back all possible examples of quote stuffing for BHP on 2013.11.08 that
lasted at least 5 seconds and contained 60 adjustments to the top of book

quoteFluctuationSym[2013.10.08;`BHP;60;5]
```

Figure 6 Quote stuffing example for BHP ($AUD) in 2013 visualised via Dashboards for Kx. The burst is comprised of over 60 alterations to the best bid over a period of 0.12 seconds.

Detecting instances of quote stuffing allows for market analysis around the time of the burst. Further analysis is used to determine market impact of the behaviour and could result in a warning or fine.

11.4.2 Price Fade

Price fade is when a trade occurs on a particular venue, and then immediately after the trade the volume disappears. This could potentially be due to traders layering the order-book, their high speed enabling such rapid cancellations. Here we look at a function that searches for price fades using market data taken from the Australian markets.
Below we look at the skeleton function for finding price fades (not all lines of codes have been included indicated by a '…'):

```
getPriceFadesData:{[dt;psym;timeThreshold;qtyThreshold;keycols]

    /- get orders and trades
... /- initial select statements have been excluded from this code block
    /- create brokerID column from a combination of other columns, then add on AJKEY
col
    /- change side to be opposite side, to join with opposite side of aggressor
indicator

    orders: `AJKEY`transactTime xasc  update AJKEY: .Q.fu[{`$"!" sv string
value x}each;(`orderside,keycols)#orders] from orders;
```

Here we have added on an AJKEY column that creates a unique key column which is a combination of `orderside`, `sym` and any other key input (e.g. `marketID`), separated by "!" and cast to symbol. It makes use of the function `.Q.fu` that is an efficient tool when a function needs to be applied across a large list that is composed by a small set of unique values. `.Q.fu` determines the distinct set of elements in the large list and their indices. It then applies the function to the small set of unique elements in the list. The results of that calculation are then applied back to the original list using the indices of each distinct element. By applying `.Q.fu` the number of times that a function is applied is reduced to be the number of distinct elements in the list.

```
    /- get window and wj on price/qty info

    w:  (tlist;tlist|(-1+trades[`nextTime]) &
timeThreshold+tlist:trades`transactTime);
```

Here we define the time windows to be used (note that `nextTime` is the next time by AJKEY from the trades data we have pulled back). Note that we only want to include information up until at most the next time a trade occurs but still inside the `timeThreshold`, hence taking the minimum of the next trade time and the `timeThreshold` + current trade time.

```
    wres: wj1[w;`AJKEY`transactTime;trades;(update `p#AJKEY from
orders;(::;`ordertime);(::;`seqNum);(::;`brokerID);(::;`orderprice);(::;`orde
rside);(::;`orderqty);(::;`orderLeavesQty))];
```

In this particular analysis we only want order information that occurs strictly within the time windows. To do this we use `wj1`, which considers orders on or after entry to the window (as opposed to `wj` which includes prevailing orders).

```
    /- add on fade indicator

    res: update fade: 0<count each brokerID from wres;
...
    /- determine full/partial fades

      update fullFade: ?[aggressorIndicator="B";(sellLeavesQty=0) and
fade;(buyLeavesQty=0) and fade], partialFade:
?[aggressorIndicator="B";(sellLeavesQty>0) and fade;(buyLeavesQty>0) and
fade] from res;

    };
```

This enables us to look at the probability of a price fade occurring:

```
/- calculates the probability of full/partial price fades occurring over time (given a
date range)

getPriceFadesProbability:{[sd;ed;psym;timeThreshold;qtyThreshold;keycols]

      /- first run price fades on weekdays

    res: getPriceFadesData[;psym;timeThreshold;qtyThreshold;keycols]  peach
dtlist where 1<(dtlist: sd+til 1+ed-sd) mod 7;
```

Above we run the `priceFades` data on a list of dates using `peach` (with the process having started with –s set to enable slaves).

```
…
      /- exit early if no price fades
    if[not count res: raze 0!' res where 0<count each res;:()];

    /- calculate probabilities of full and partial fades

    select probFullFade: 100*sum[fullFade]%count i, probPartialFades:
100*sum[partialFade]%count i, trades: count i by bkt xbar
transactTime.minute from     res

    };
```

Here we can determine full takes (i.e. when the entire quantity has been filled)

```
getPriceFades:{[dt;psym;timeThreshold;qtyThreshold;keycols]

    res: getPriceFadesData[dt;psym;timeThreshold;qtyThreshold;keycols];

    res: 0! select by transactTime, orderSeqNum from ungroup select
transactTime,orderSeqNum:seqNum, price, brokerID, orderqty, buyLeavesQty,
sellLeavesQty, orderprice,orderside,aggressorIndicator from res;

      /- classify full takes as having no leaves quantity remaining

    update fullTake: ?[aggressorIndicator="S"; buyLeavesQty=0;
sellLeavesQty=0] from select from res

    };
```

This enables us to profile different brokers based on partial fades, full fades, finding the average order quantities for each.

```
/- profiling different brokers

      getPriceFadesAggregated:
      {[dt;psym;timeThreshold;qtyThreshold;keycols]

    /- pull out price fades

    fades: getPriceFades[dt;psym;timeThreshold;qtyThreshold;keycols];

      /- aggregate the partial/full fades by brokers

    select partialFades: sum  not fullTake, fullFades: sum fullTake,
bothFades: count i, avgFullTakes: avg fullTake*orderqty,  avgPartialTakes:
avg orderqty*not fullTake, avgBoth: avg orderqty by brokerID from fades

 };
```

It is interesting to look at the price fade time profile over a range of dates

```
/- for profiling over time, range of dates

getPriceFadesTimeProfileDates:{[sd;ed;psym;timeThreshold;qtyThreshold;keycols
;bkt]

    /- pull out price fades

     fades:getPriceFades[;psym;timeThreshold;qtyThreshold;keycols] peach
dtlist where 1<(dtlist: sd+til 1+ed-sd) mod 7;
```

```
...

      /- time profile over bkt sized minute buckets

    select partialFades: sum  not fullTake, fullFades: sum fullTake,
bothFades: count i by bkt xbar transactTime.minute from fades

 };
```

Examples

Sample

Outputs:

Sample 1: full price fade, time threshold set to 00:00:00.1, min order qty set to 100

```
transactTime       | 2013.10.08D10:17:10.888910900
sym                | `SYMA
marketSegmentID    | `MKTA
price              | 1.83
seqNum             | 451421 451422
qty                | 8f
buyLeavesQty       | 0f
sellLeavesQty      | 376f
aggressorIndicator| "S"
AJKEY              | `S!SYMA!MKTA
nextTime           | 2013.10.07D23:17:11.332014400
ordertime          | 2013.10.07D23:17:10.946351200
2013.10.07D23:17:10.946492600
brokerID           | ,`BROKER1!ACCT1!SECACCT1
orderprice         | 1.85 1.855
orderside          | "SS"
orderqty           | 6972 12138f
orderLeavesQty     | 0 0f
fade               | 1b
fullFade           | 1b
partialFade        | 0b
```

Sample 2: partial fade, time threshold set to 00:00:00.1, min order qty set to 100

```
transactTime       | 2013.10.08D12:03:05.133341400
sym                | `SYMB
marketSegmentID    | `MKTB
price              | 1.085
seqNum             | 2204182 2204184
qty                | 47f
buyLeavesQty       | 0f
sellLeavesQty      | 7254f
aggressorIndicator| "B"
AJKEY              | `B!SYMB!MKTB
nextTime           | 2013.10.08D01:03:05.164701700
ordertime          | 2013.10.08D01:03:05.146682300
2013.10.08D01:03:05.148648600
brokerID           | `BROKER1!ACCT1!SECACCT1`BROKER2!ACCT2!SECACCT2
orderprice         | 1.08 1.08
orderside          | "BB"
orderqty           | 604 7342f
orderLeavesQty     | 0 0f
fade               | 1b
fullFade           | 0b
partialFade        | 1b
```

Note on timing:

All the examples and sample outputs shown here are being run on a regular laptop computer using the 32 bit version (specs: 2.4 GHz Intel Core i5, Memory: 8 GB 1333 MHz DDR3), and even then the timings are relatively fast. It must be noted that the price fades functions are expensive by nature, e.g. implementing a window join – which involves a lot of heavy processing. Sample timing below:

```
KDB+ 3.1 2014.03.27 Copyright (C) 1993-2014 Kx Systems
m32/ 4()core 8192MB ryansparks ryans-macbook-pro.local 192.168.2.30 NONEXPIRE

q) \t  getPriceFadesData[2013.10.07;`;0D00:00:00.1;100;`sym`marketSegmentID]
6510j
```

This clearly highlights the speed and succinctness with which kdb+ can be used to monitor and analyse high frequency trading patterns.

Example 1: probability of price fades occurring, under 00:00:00.001 second threshold

The probability of full fades occurring here are shown to be around 10% throughout the trading day, with partial fades much more likely to occur.

Figure 7 Probability of price fades, 0.001 second time threshold. Full price fade in red, partial in blue

Example 2: probability of price fades occurring, under 00:00:00.1 second threshold

The probability of full fades is close to 12% throughout most of the trading day, with partial fades only about 30% as likely.

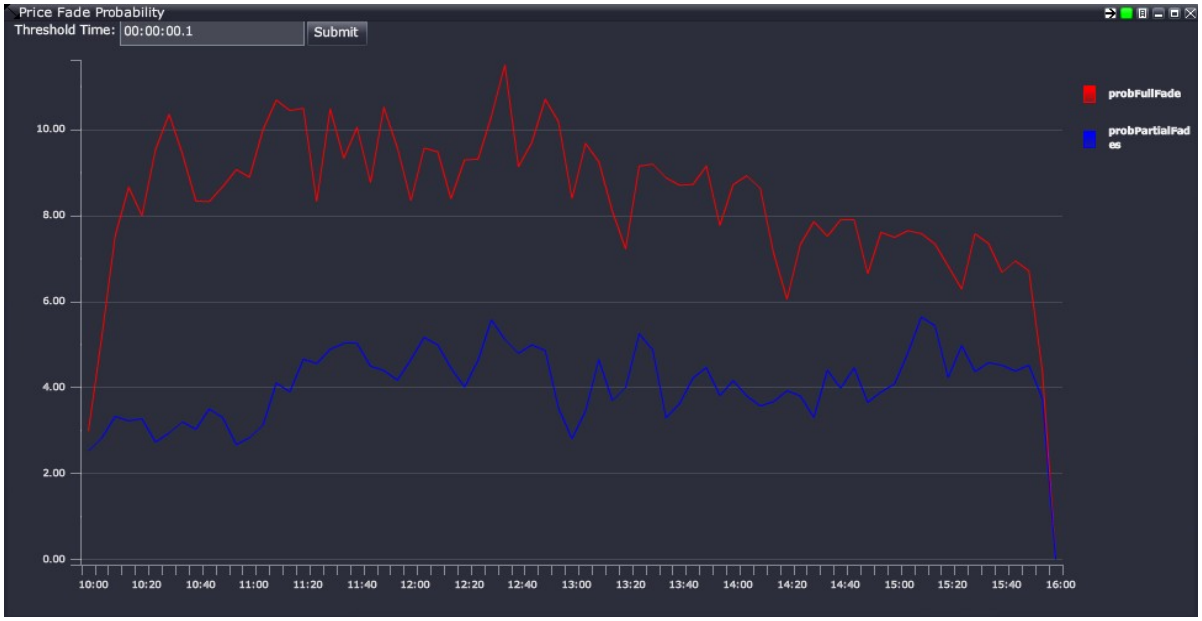


Figure 8 Probability of price fades, 0.1 second time threshold. Full price fade in red, partial in blue

Example 3: probability of price fades occurring, under 00:00:01 second threshold

With a much higher time threshold, full fades are about 14% likely to occur. Interestingly, partial fades occur at around 7%, relatively higher than the smaller time thresholds

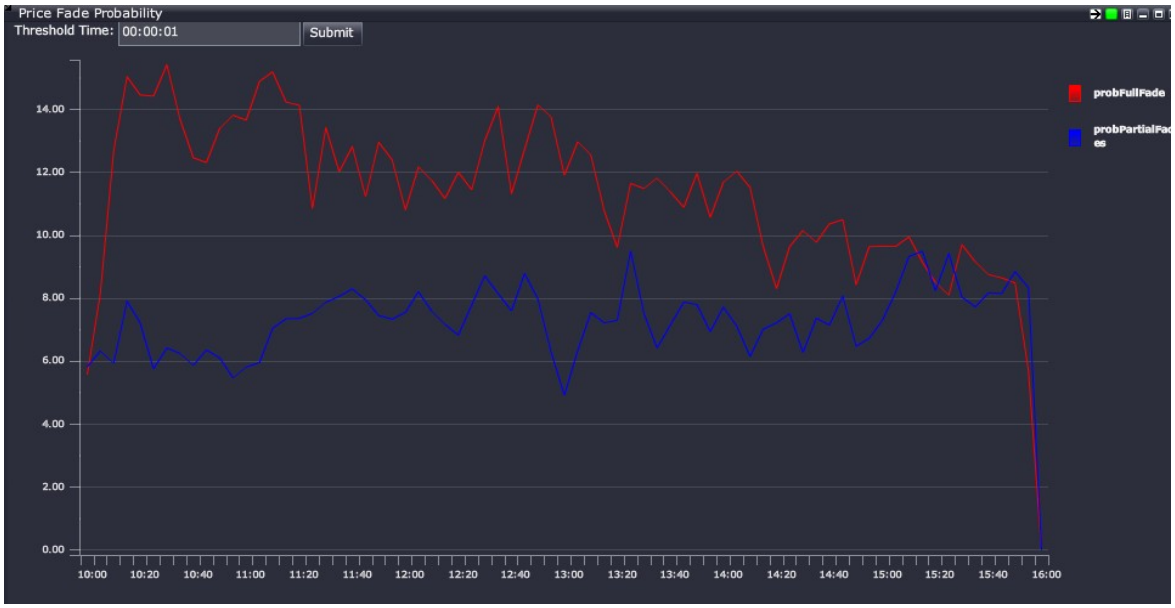


Figure 9 Probability of price fades, 1 second time threshold

Example 4: volume of full vs. partial fades throughout a trading day, using 00:00:00.1 time threshold

The highest volume of fades here tend to occur around lunch time and towards the end of the day, interestingly both full and partial fades follow a very similar trend.

Figure 10 Volume of price fades, 0.1 second time threshold

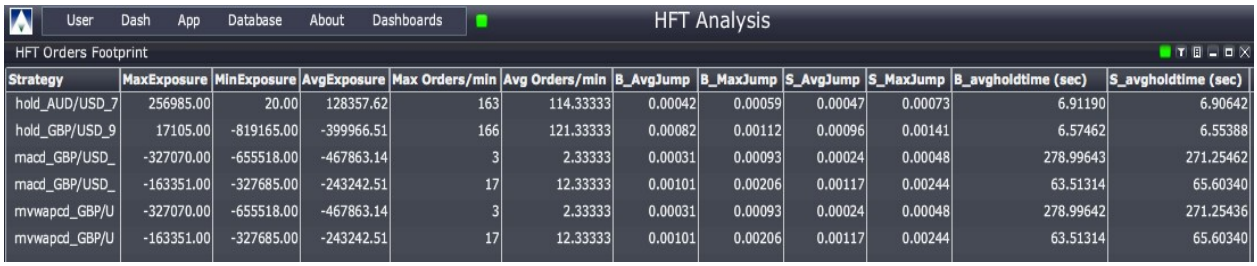
Example 5: average size of full fades, using 1 second time threshold

We see a relatively large number of mid sized price fades occurring.

```
fadeSize| fadeCount
--------| ---------
0       | 31070
50      | 12139
100     | 34775
500     | 13856
1000    | 19926
10000   | 2575
100000  | 113
```

Example studies of real time HFT analysis:

Here we examine some of the footprints a HFT exhibits.



Strategy	MaxExposure	MinExposure	AvgExposure	Max Orders/min	Avg Orders/min	B_AvgJump	B_MaxJump	S_AvgJump	S_MaxJump	B_avgholdtime (sec)	S_avgholdtime (sec)
hold_AUD/USD_7	256985.00	20.00	128357.62	163	114.33333	0.00042	0.00059	0.00047	0.00073	6.91190	6.90642
hold_GBP/USD_9	17105.00	-819165.00	-399966.51	166	121.33333	0.00082	0.00112	0.00096	0.00141	6.57462	6.55388
macd_GBP/USD_	-327070.00	-655518.00	-467863.14	3	2.33333	0.00031	0.00093	0.00024	0.00048	278.99643	271.25462
macd_GBP/USD_	-163351.00	-327685.00	-243242.51	17	12.33333	0.00101	0.00206	0.00117	0.00244	63.51314	65.60340
mvwapcd_GBP/U	-327070.00	-655518.00	-467863.14	3	2.33333	0.00031	0.00093	0.00024	0.00048	278.99642	271.25436
mvwapcd_GBP/U	-163351.00	-327685.00	-243242.51	17	12.33333	0.00101	0.00206	0.00117	0.00244	63.51314	65.60340

Figure 11 HFT and Algo profiling, whereby strategies become evident through attributes and metrics such as order rates, trade rates, holding times, buy/sell impact on the book etc. This could include Order to Trade ratios.

11.5 CONCLUSION

The focus of market surveillance is to maintain integrity within the market by ensuring there is a fair and safe environment for all participants. High frequency trading has the potential to make the market unsafe, and as such it is important to be able to identify and label high frequency traders. By specifically analysing their trading activity, unsavoury and predatory techniques can be better regulated and stamped out by the regulator and exchange alike. However, legitimate high frequency trading brokers need to be reassured that they will not be persecuted if they operate ethically and within their legal bounds. kdb+ provides a perfect platform for doing this, with its flexibility and high speed allowing real time and historical analysis of these strategies to be developed quickly and dynamically as the HFT world evolves. The analysis can be used to detect algorithmic induced price crashes, and therefore give exchanges the power and foresight to stop possible crashes before too much damage is done. Some simple real market analysis with kdb+ indicates there are instances of unsavoury HFT practices occurring within the Australian market. The fundamental analysis found that the probability of a full price fade occurring is around 7 to 15% throughout a trading day. A standard personal laptop can analyse 10 million orders and 1 million trades for full and partial price fade information in under 7 seconds using the 32 bit kdb+ version, giving an indication of how powerful kdb+ is for big data time series analysis. Having all of this done in around 10 lines of (relatively verbose for q) code shows how it is the perfect technology for HFT analysis. As the effects of such HFT practices are still not completely understood, this highlights the need for more stringent regulation, testing and research.

In September of 2013 the Australian market regulator ASIC upgraded their market surveillance system to a kdb+ solution. Greg Yanco, a senior executive leader at ASIC, believes that the upgrade has formed a faster, more flexible environment for their market analysts (fixglobal, 2014). The potential for storing and selecting data with kdb+ has made this solution ideal for innovating techniques and developing new ideas for the surveillance industry as whole, making ASIC a world leader in surveillance technology.

All tests performed using kdb+ version 3.1 (2014.02.08)

APPENDIX A – Kx SOLUTIONS

A detailed list of the delta products referred to and demonstrated in this paper can be viewed at http://www.kx.com.

APPENDIX B – MARKET DATA

The data analysed throughout the paper is based on scrambled Australian Market data captured from the ASX, CHIX and XSFE markets for the purpose of surveillance. Below is a brief description and a simplified `meta` of tables that are referred to in sections 3 and 4.

- **dxOrder:** order table, containing approximately 10 million rows per day. This table contains all order transactions at a broker, account and client level. The types of order messages contained in the table include new, amend, cancel and restated messages.

```
meta dxOrder
c                  | t f a
-------------------| -----
transactTime       | p         transaction time from exchange
sym                | s         sym
marketSegmentID    | s         the market it traded on
seqNum             | j         exchange sequence number
price              | f         order price
qty                | f         original order quantity
side               | c         order side
leavesQty          | f         quantity remaining on order
brokerID           | s         relevant broker
account            | C         account  ID
secondaryAccountID| C          secondary account ID
```

- **dxTrade:** trade table, containing approximately 1 millions rows per day. Here is a basic meta of the table:

```
meta dxTrade
c                  | t f a
-------------------| -----
transactTime       | p         transaction time from exchange
sym                | s         sym
marketSegmentID    | s         market traded on
price              | f         traded price
seqNum             | j         exchange sequence number
qty                | f         traded quantity
buyLeavesQty       | f         quantity remaining on buy order
sellLeavesQty      | f         quantity remaining on sell order
buyBrokerID        | s         broker identifier on the buy order
buyAccount         | C         account identifier on the buy order
sellBrokerID       | s         broker identifier on the sell order
sellAccount        | C         account identifier on the sell order
aggressorIndicator| c          aggressor indicator
```

Bibliography

ASIC. (2013). *Report 331 - Dark liquidity and high-frequency trading.* Australian Securities and Investments Commission.

Comstock, C. (2010). *First High Frequency Firm is Fined for Quote Stuffing and Manipulation.* Business Insider.

Credit Suisse. (2012). *High Frequency Trading – Measurement, Detection and Response.*

Finextra. (2014). *ASIC hits out at 'Flash Boys' hysteria.*

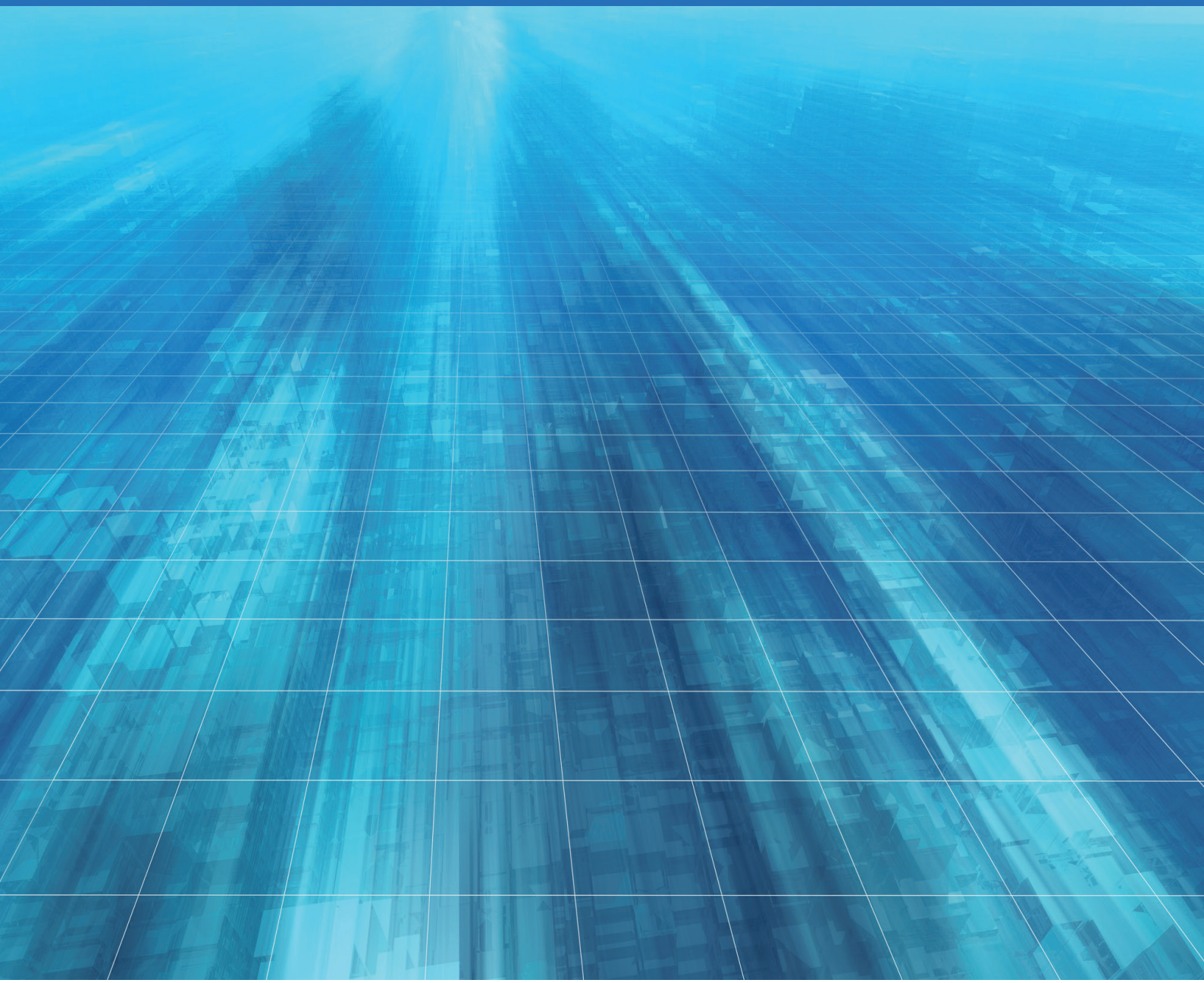
fixglobal. (2014). *The Challenge Of Global Market Surveillance.*

Knowledge@Wharton. (2009). *The Impact of High-frequency Trading: Manipulation, Distortion or a Better-functioning Market?*

Sydney Morning Herald. (2014). *High-speed trading costs investors $2b, say industry super.*

University of Pennsylvania. (2009). *The Impact of High-frequency Trading: Manipulation, Distoring or a Better-functioning Market?* USA.

kx | it's about time



Compendium of Technical White Papers from **Kx**