NO STINKING LOOPS

Treetable: a case-study in q

Stevan Apter

This article is the first in an occasional column, *No Stinking Loops*. Stevan Apter is one of the programmers Jeffry Borror referred to as "the q gods" in his textbook *q for Mortals*. The world of q programming has so far been largely hidden behind corporate non-disclosure contracts. *Vector* is glad to see it opening and proud to be publishing this. *Ed.*

0. Introduction

A treetable is a table with four additional properties.

Firstly, the records of the table are related hierarchically. Thus, a record may have one or more child-records, which may in turn have children. If a record has a parent, it has exactly one. A record without a parent is called a root record. A record without any children is called a leaf record. A record with children is called a node record.

Secondly, it is possible to *drill down* into a treetable. If a record is a parent, then some of its columns may be rollups of its child-records. By drilling down into a parent-record, it is possible to inspect the elements which are aggregated in the parent. All rollups are performed on the leaves of the tree rather than on the immediate children. This means that tree-construction can be 'lazy': not all intermediate rollups from parent to leaves need exist.

Thirdly, treetables have *state*. If the user drills down into the tree along a particular path, then closes a node along that path, the records on that path become invisible. If the user re-opens that parent, then the nodes along that path will become visible if they were visible before the parent was closed. In other words, closing an open parent does not destroy the visibility state of its children.

Fourthly, a treetable is naturally sorted in a way that is an extension of ordinary table sort. Intuitively, the sort of a treetable is a structure-preserving sort of the 'blocks' out of which it is composed. The sort is *structure-preserving* because the parent-child relation between records is preserved even though record-order is not. I've included an explanation of how such a multi-column sort works.

The treetable is a natural candidate for a control in a non-procedural data-driven GUI. K and q have a long tradition of such GUIs, stretching back to A+ and the native K3 GUI. The examples in this paper are abstracted from an implementation of a GUI recently developed for q. But the case-study is meant to stand alone, as an exercise in pure data design. In a future instalment, I hope to show how such designs are smoothly integrated into a data-driven GUI.

1. Lists, dictionaries, tables, keytables

This section contains the necessary background on q's collection data-types.

A list is a collection indexed by position:

```
q)l:10 20 30

q)l 2 0
30 10

q)l?30 10
2 0
```

A dictionary is a collection indexed by a q object:

```
q)d:`a`b`c!10 20 30

q)d`c`a
30 10

q)d?30 20
`c`b

q)key d
`a`b`c

q)value d
10 20 30
```

The q object is usually a symbol (a name) but need not be. For example:

```
q)e:(10 20;30 40 50;70 80)!`a`b`c
q)e(30 40 50;10 20)
`b`a
```

A dictionary is a map from a list of elements (the key) to a list of elements (the value). The key and the value must have the same count. Moreover, the key should not contain duplicates. Although q does not enforce key-uniqueness, dictionaries containing duplicate keys may not behave as you'd expect.

Atomic functions penetrate both lists and dictionaries:

```
q)l+1
11 21 31

q)d+1
a| 11
b| 21
c| 31
```

A table is a list of dictionaries, or *records*, all of which have the same key. For example:

```
q)t:(d;d+1;d+2;d+3)
q)t
a  b  c
--------
10 20 30
11 21 31
12 22 32
13 23 33
```

A list of key-dissimilar dictionaries is not a table:

```
q)(`a`b!10 20;`b`c`d!30 40 50)
`a`b!10 20
`b`c`d!30 40 50
```

Since tables are lists, we can index them positionally:

```
q)t 1
a| 11
b| 21
c| 31
```

The *transpose* of a table is a dictionary whose values are lists:

```
q)flip t
a| 10 11 12 13
b| 20 21 22 23
c| 30 31 32 33
```

The *flip* of a dictionary of equal-length lists is a table:

```
q)t~flip flip t
1b
```

q has a compact notation for constructing tables:

```
q)u:([]a:10 11 12 13;b:20 21 22 23;c:30 31 32 33)
q)t~u
1b
```

A table can therefore be constructed in three ways:

as a list of dictionaries

as the flip of a dictionary of vectors

using table-notation

A keytable is a dictionary in which the key and the value are both tables. For example:

```
q)k:([]f:`a`a`b;g:1 2 1)
q)v:([]a:10 20 30;b:40 50 60;c:70 80 90)
q)a:k!v
q)a
f g| a  b  c
---| --------
a 1| 10 40 70
a 2| 20 50 80
b 1| 30 60 90
```

Since keytables are dictionaries, they are indexed by key:

```
q)a(`a;1)
a| 10
b| 40
c| 70

q)a((`a;1);(`a;2))
a  b  c
--------
10 40 70
20 50 80
```

Since keytables are dictionaries, they can be split into key and value:

```
q)key a
f g
---
a 1
a 2
b 1

q)value a
a  b  c
--------
10 40 70
20 50 80
30 60 90
```

A keytable can also be defined using q table-notation:

```
q)a~([f:`a`a`b;g:1 2 1]a:10 20 30;b:40 50 60;c:70 80 90)
1b
```

The data universe of q can be summarised as follows: There are atoms and lists. Dictionaries map lists to lists. Tables are lists of dictionaries and keytables are dictionaries which map tables to tables.

2. Trees

We can represent a tree as a list of paths. For each element of the tree there is a corresponding path to that element:

```
tree            path
----            ----
A               A
 B              A B
  C             A B C
  D             A B D
 E              A E
  F             A E F
   G            A E F G
   H            A E F H
   I            A E F I
```

From the path of an element e we can easily compute the parent: the parent of `e` is `enlist e` if the path is a singleton, else the parent is the `drop` of the last element of the path.

While the path-list representation is intuitive, it can be clumsy to work with. For example, to find the children of `A` we need to search the path-list for all 2-element lists which have `A` as their first element. To find the children of `A B` we need to find all 3-element lists which have `A B` as their first two elements.

We can represent the parent-child relation explicitly, as a table `PC`:

```
parent  child
------  -----
A       A
A       B
B       C
B       D
A       E
E       F
F       G
F       H
F       I
```

Then:

```
q)select child from PC where parent=`A
child
-----
B
E

q)select parent from PC where child=`F
parent
------
E
```

We can represent the relation as a parent-vector:

```
p:0 0 1 1 0 4 5 5 5
```

That is:

```
i       tree            p
-       ----            -
0       A               0
1        B              0
2         C             1
3         D             1
4        E              0
5         F             4
6           G           5
7           H           5
8           I           5
```

`p[i]` is the index of the parent of the `i` th element of the tree.

To find the children of any element `e` in the tree, search `p` for occurrences of `e`'s index:

```
q)where p=5
6 7 8
```

To find the root of any element `e`, repeatedly index `p`, starting with `e`:

```
q)p 6
5
q)p 5
4
q)p 4
0
q)p 0
0
```

To find the root of `e` in one step, reduce `p` over `e`:

```
q)p over 6
0
```

`p` is applied repeatedly to the previous result until the result is the same twice in a row. This is why it is convenient to treat the root as self-parenting.

To find the path from `e` to the root in one step, scan `p` over `e`:

```
q)p scan 6
6 5 4 0
```

To find the paths of all elements of the tree, scan `p` over each of 0 ... `count[p]-1` :

```
q)i:(p scan)each til count p
q)i
,0
1 0
2 1 0
3 1 0
4 0
5 4 0
6 5 4 0
7 5 4 0
8 5 4 0
```

To find the leaf-elements of the tree, discard every element which is a parent:

```
q)l:til[count p]except p
q)l
2 3 6 7 8
```

We can use `p` to aggregate data associated with the tree. For example, suppose that the five leaf-elements have the following data:

```
q)d:count[p]#0
q)d[l]:l*10
q)d
0 0 20 30 0 0 60 70 80
```

Now to sum `d` to the root, amend a zero-vector with `+` at `i` , the effect of which is to accumulate sums on all paths:

```
q)@[count[d]#0;i;+;d]
260 50 20 30 210 210 60 70 80
```

Think of it this way: where `r` is the result, `r k` is `sum d i k` . In other words, the result at each node of the tree is the sum of that node's descendants.

From the parent-vector representation and a list of elements, we can compute the path-list:

```
q)e:`A`B`C`D`E`F`G`H`I
q)n:reverse each e i
q)n
,`A
`A`B
`A`B`C
`A`B`D
`A`E
`A`E`F
`A`E`F`G
`A`E`F`H
`A`E`F`I
```

And from the path-list representation we can compute the parent-vector: drop the last element of each path whose count is greater than 1 and find each truncated path in the path-list:

```
q)n?neg[1<count each n]_'n
0 0 1 1 0 4 5 5 5
```

From `p` and `e` it is also easy to derive the parent-child table:

```
q)PC:([]parent:e p;child:e)
q)PC
parent child
------------
A      A
A      B
B      C
B      D
A      E
E      F
F      G
F      H
F      I
```

And vice-versa:

```
q)e?PC.parent
0 0 1 1 0 4 5 5 5
```

3. Treetables

One way to think about the treetable is that it is a keytable whose records are related by the parent-child relation.

A record is either a leaf or a parent. A parent record is the rollup of its children.

A treetable has a single grand-total record, the root of the tree.

The parent-child relation is constructed from an underlying table `T`. The records of `T` are precisely the leaf-records of the treetable. Nothing more is required of `T` except that it be a table, but in practice all suitable candidates for `T` will conform to the following condition: `T` will contain one or more columns which are suitable to group by, and one or more columns which are suitable to aggregate.

For example, the following table satisfies that condition:

```
A B C v  w
----------
a f n 12 x
a f o 10 y
a f p 1  z
a f q 90 w
a g n 73 x
a g o 90 y
…
```

`A`, `B`, and `C` are suitable to group by, and `v` and `w` are suitable to aggregate. But it is worthwhile emphasising that this distinction is entirely arbitrary, and that nothing in the algorithm requires that columns of either type have any special properties. It would be silly to group on a column most of whose values are different, but the algorithm doesn't preclude that. And in this example, although `w` is a column of symbols, it can be aggregated as long as its rollup function satisfies the condition that it takes list input and returns an atom.

Let's look at one possible treetable based on `T`. The grouping columns are `A`, `B`, and `C`. Order matters. `T` grouped by `A B C` is different from `T` grouped by `B A C`. The rollup columns are `v` and `w`, and the rollup functions for those columns are `sum`, `nul` (explained below), and `count`.

A single column may be aggregated more than once. In the example below, we aggregate `v` with `sum` and `count`.

A treetable based on that scheme is `R1`:

```
n_         | A B C counts v     w
-----------| --------------------
`symbol$()|        1000   52015
,`a        | a      224    12054
,`b        | b      200    11173
,`c        | c      192    10290
,`d        | d      192    8136
,`e        | e      192    10362
```

Here we can see that `R1` is a keytable. The key of `R1` is the column `n_`, a path-list. The first record of `R1` is the grand-total of `T`. We can see that `T` has 1000 records and that column `v` sums to 52015. The aggregations of column `w` are null.

We can also see by examining the remaining records that `R1` contains a single level of aggregation based on distinct values of column `A`.

We can also see nulls in `R1`: the `A` column of the first record, and all of column `w`. In the examples used in this paper, null means *cannot aggregate this group*.

Now let's drill down on the record where `A=`a`, giving us the table `R2`:

```
n_         | A B C counts v     w
-----------| --------------------
`symbol$()|        1000   52015
,`a        | a      224    12054
`a`f       | a f    28     791
`a`g       | a g    28     2072
`a`h       | a h    28     2058
`a`i       | a i    28     1967
`a`j       | a j    28     1078
`a`k       | a k    28     1645
`a`l       | a l    28     1484
`a`m       | a m    28     959
,`b        | b      200    11173
,`c        | c      192    10290
,`d        | d      192    8136
,`e        | e      192    10362
```

One way to think about treetables like `R1` and `R2` is that they are constructed out of sub-tables. These 'blocks' are computed independently from `T`, then stitched together in the right order. This is the pattern followed by the algorithm described below.

Let's begin by identifying the parameters.

The first parameter is `T`, the underlying table of unaggregated records.

The second parameter is a list of the grouping columns:

```
q)G:`A`B`C
```

The third parameter is a dictionary of rollup functions:

```
q)A:`counts`v`w!((sum;`n);(sum;`v);(nul;`w))
q)A
counts| count                                  `v
v     | sum                                    `v
w     | {first$[1=count distinct x,();x;0#x]} `w
```

The key of `A` is a list of names of the aggregated columns in the treetable. The value of `A` is a list of pairs of the form `(f;c)`, where `f` is an aggregator and `c` is a column in `T`.

`count` and `sum` are primitive aggregators of q: `sum` is `+/` and `count` returns the number of elements in a list. `nul` serves as our general default aggregator: given a list `x`, return the first element of `x` if `x` is all duplicates, else return the null of `x`. Nulls in treetables are always the result of aggregation by `nul`.

The fourth and final parameter is a package of information which represents the "drill-down state" of the treetable to be computed. For R1, this state is the keytable P1:

```
n                      | v
-----------------------| -
(`symbol$())!`symbol$()| 1
```

and for `R2` it is the keytable `P2`:

```
n                      | v
-----------------------| -
(`symbol$())!`symbol$()| 1
(,`A)!,`a              | 1
```

The state is a keytable where the key `n` is a list of dictionaries, each of which functions as an instruction to the algorithm to compute a specific sub-table block of the treetable. The meaning of `v` is described below in the section on state.

For example, the grand-total block is arbitrarily represented by the unique empty dictionary:

```
(`symbol$())!`symbol$()
```

whose key and value both consist of the empty symbol list. This dictionary will be interpreted as the instruction to select all records from the underlying table `T` and aggregate them by distinct values of the first element of `G`, which in this example is `` `A ``.

The `R2` block of aggregated `A=`a` values consists of the dictionary:

```
A| a
```

This will be interpreted as the instruction to select records from `T` where `A=`a` and aggregate them by distinct values of the second element `G`, which in this example is `` `B ``.

Finally, let's look at a treetable `R4` which has been drilled down to the leaves along one of the paths:

```
n_         | A B C counts v      w
-----------| --------------------
`symbol$()|        1000   479131
,`a        | a      224    106670
`a`f       | a f    28     13952
`a`f`n     | a f n  7      2867   x
`a`f`n`0   | a f n  908    908    x
`a`f`n`1   | a f n  256    256    x
`a`f`n`2   | a f n  401    401    x
`a`f`n`3   | a f n  288    288    x
`a`f`n`4   | a f n  543    543    x
`a`f`n`5   | a f n  258    258    x
`a`f`n`6   | a f n  213    213    x
`a`f`o     | a f o  7      3707   y
`a`f`p     | a f p  7      3640   z
`a`f`q     | a f q  7      3738   w
`a`g       | a g    28     14948
`a`h       | a h    28     12190
`a`i       | a i    28     13535
`a`j       | a j    28     13835
`a`k       | a k    28     12945
`a`l       | a l    28     13643
…
```

We append a unique identifier to the key of each leaf. This preserves `n_` as a valid key.

The instruction table for `R4` is `P4`:

```
n                       | v
------------------------| -
(`symbol$())!`symbol$()| 1
(,`A)!,`a               | 1
`A`B!`a`f               | 1
`A`B`C!`a`f`n           | 1
```

`R4` has the following blocks as constituents:

- the grand-total block (root)
- the `A=`a` block
- the `A=`a, B=`f` block
- the `A=`a, B=`f, C=`n` block (leaves)

Concentrating just on the value parts of the blocks, let's see how we would generate those using the native query language of q.

To compute the grand-total block:

```
q)flip enlist each exec nul A,nul B,nul C,count v,sum v,nul w from T
A B C v    v1     w
-------------------
           1000 479131
```

To compute the first subtotal level:

```
q)0!select nul B,nul C,counts:count v,sum v,nul w by A from T
A B C counts v      w
---------------------
a     224    106670
b     200    100048
c     192    90541
d     192    92853
e     192    89019
```

To compute the `A=`a` block:

```
q)0!select nul C,counts:count v,sum v,nul w by A,B from T where A=`a
A B C counts v     w
--------------------
a f   28     13952
a g   28     14948
a h   28     12190
a i   28     13535
a j   28     13835
a k   28     12945
a l   28     13643
a m   28     11622
```

To compute the `A=`a, B=`f` block:

```
q)0!select counts:count v,sum v,nul w by A,B,C from T where A=`a,B=`f
A B C counts v    w
-------------------
a f n 7      2867 x
a f o 7      3707 y
a f p 7      3640 z
a f q 7      3738 w
```

And finally, to compute the block containing the leaves:

```
q)0!select A,B,C,counts:1,v,w from T where A=`a,B=`f,C=`n
A B C counts v   w
------------------
a f n 1      908 x
a f n 1      256 x
a f n 1      401 x
a f n 1      288 x
a f n 1      543 x
a f n 1      258 x
a f n 1      213 x
```

4. Construction

Now we know what a treetable is, and have an intuitive grasp of what its parts are, how they are related, and how those parts are computed. The next step is to explain the q code which implements those ideas. My advice is that the reader get a q session, load the associated script[1], and experiment by reading along and executing (and varying!) bits of code. All the examples used in this paper are defined in that script.

Rather than write a line-by-line commentary on the implementation, I've chosen to focus on the concepts which drive that implementation, and on a few of the knottier parts of the code.

An indispensable companion in this (the reader's) task is Jeffry Borror's splendid book, *q for Mortals*. There are a few 'dangerous curves' ahead. In particular, I recommend close study of the chapter in Jeffrey's book on Functional Forms.

The four parameters of treetable construction are:

`T` the underlying table
`G` a list of group columns
`P` the path table
`A` the rollup dictionary

The construction function takes `T`, `G`, `P`, and `A` and returns `R`, the treetable:

```
construct:{[t;g;p;a]1!`n_ xasc root[t;g;a]block[t;g;a]/visible p}
```

`construct` uses three subfunctions:

`visible` determine which paths are visible
`root` construct the root block
`block` construct non-root blocks

The form of the `construct` function is:

```
.. r0 f/s
```

`r0` is the initial state and `s` is a list of arguments to the dyadic function `f`. Note that the block function takes five arguments, but in this context is applied as a dyad. In q, we say that `block` is *projected* on its first three arguments `t`, `g`, and `a`. The first three arguments are fixed and the remaining two argument positions are open. So `block[t;g;a]` is a dyad.

Suppose `s` has `n` elements. Then:

```
r1:f[r0;s 0]
r2:f[r1;s 1]
:
rn:f[rn-1;s n-1]
```

In this case, `r0` is the root block of the treetable and `s` is the result of applying `visible` to the path table `p`. For now, all we need to know is that this result is a list of instructions, for example:

```
(`symbol$())!`symbol$()
(,`A)!,`a
`A`B!`a`f
```

So in this example, the block function `f` will be applied three times: first to the root block and the first instruction; then to the result of that application and the second instruction; and finally to the result of that application and the third instruction.

The result is a table with the structure:

```
root block
A=`a block
A=`a, B=`f block
```

The `construct` function then up-sorts the result by `n_` and makes it the key:

```
1!`n_ xasc ..
```

This is necessary because the blocks have to be recursively interleaved. For example, the `A=`a, B=`f` block must appear in the treetable immediately after the record in the `A=`a` block where `B=`f`:

```
n_         | A B C counts v      w
-----------| ---------------------
`symbol$()|       1000   479131
,`a        | a     224    106670
`a`f       | a f   28     13952
`a`f`n     | a f n 7      2867   x
`a`f`o     | a f o 7      3707   y
`a`f`p     | a f p 7      3640   z
`a`f`q     | a f q 7      3738   w
`a`g       | a g   28     14948
`a`h       | a h   28     12190
…
```

Sorting on `n_` is a fast non-recursive method for interleaving records from the different blocks.

`construct` uses `over` to produce a single table, where successive blocks are appended to the initial root table. This method destroys structural information about the blocks. That is, we have a single table as the result rather than a list of blocks.

There is an alternative method which constructs the blocks in parallel using `peach` instead of `over`. Assume q is started with slaves (e.g. `q -s 4`). Then:

```
pconstruct:{[t;g;p;a]1!`n_ xasc root[t;g;a],raze pblock[t;g;a]peach visible p}

pblock:{[t;g;a;p]
 f:$[g~key p;leaf;node g(`,g)?last key p];
 (`n_,g)xcols f[t;g;a;p]}
```

We'll now look more closely at the root and block functions. The visible function is discussed in section 5 below. A few auxiliary functions mentioned in the text are not discussed.

The root function is:

```
root:{[t;g;a]
 a[g]:nul,'g;
 (`n_,g)xcols node_[g]flip enlist each?[t;();();a]}
```

Recall from the previous section that the root block is computed with the expression:

```
flip enlist each exec nul A, nul B, nul C, counts:count v, sum v, nul w from T
```

We can use q's native parsing primitive to see the underlying functional form of the query part of this expression:

```
q)parse "exec nul A, nul B, nul C, counts:count v, sum v, nul w from T"
?
`T
()
()
`A`B`C`counts`v`w!((`nul;`A);(`nul;`B);(`nul;`C);(#:;`v);(sum;`v);(`nul;`w))
```

The functional form of exec is:

```
?[t;();();a]
```

where a is the rollup dictionary constructed in the first two lines of the root function. In the case where every element of `a` is a rollup, this expression returns a dictionary of atoms. To get our one-record table, we therefore enlist each atom and flip the result. This table is now passed to the `node_` function, which adds the `n_` column which will become the key of our root block. The result is reordered to put `n_` and the grouping columns at the front.

The `block` function doesn't do much: it calls the `leaf` function if the key of the instruction contains all the grouping columns, else it calls the `node` function with by-clause `b` = the next grouping column:

```
block:{[t;g;a;r;p]
 f:$[g~key p;leaf;node g(`,g)?last key p];
 r,(`n_,g)xcols f[t;g;a;p]}
```

As the final step, it pushes the key and the grouping columns out to the front of the query result and appends this to the treetable `r` computed to this point.

The node function is:

```
node:{[b;t;g;a;p]
 c:constraint p;
 a[h]:first,'h:(i:g?b)#g;
 a[h]:nul,'h:(1+i)_g;
 node_[g]0!?[t;c;enlist[b]!enlist b;a]}
```

Recall again from the previous section how we compute a block which is neither a leaf nor a root:

```
select nul C, counts:count v, sum v, nul w by A,B from T where A=`a
```

The functional form of this query is:

```
?
`T
,,(=;`A;,`a)
`A`B!`A`B
`C`counts`v`w!((`nul;`C);(#:;`v);(sum;`v);(`nul;`w))
```

In an expression of the form:

```
?[t;c;b;a]
```

`c` is the constraint, or 'where' clause (where `A=`a`), `b` is the grouping, or 'by' clause (by `A,B`), and `a` is the rollup dictionary.

The first line of the function constructs the 'where' clause `c` from the instruction `p`:

```
constraint:{[p]flip(=;key p;flip enlist value p)}
```

For example,

```
q)p
A | a
B | f
C | n

q)constraint p
= `A ,`a
= `B ,`f
= `C ,`n
```

The next two lines construct the 'by' clause from `A` and the group column vector `g`.

In the last line, the constructed query is evaluated, de-keyed, and passed through the `node_` function, which adds the `n_` column to the table. The columns are re-ordered in the `block` function, which calls `leaf` and `node`.

The leaf function has a similar form:

```
leaf:{[t;g;a;p]
 c:constraint p;
 a:last each a;
 a[g]:g;
 leaf_[g]0!?[t;c;0b;a]}
```

Again, the first three lines construct the arguments to the functional form of the leaf query, and the resulting table is de-keyed and passed through the leaf_ function, which adds the n_ column to the result.

5. State

The treetable is intended for interactive use as the data-structure backing a GUI control. The user of the control clicks on a record to open or close that record. Opening a record `r` in the control reveals the records which are children of `r`. Closing `r` conceals the children of `r`.

If a child `c` of `r` is open, and then `r` is closed and re-opened, then `c`'s state must be restored. Therefore we must keep track of the state of the treetable. We do this by associating the instruction for a block with a boolean value. The value is `1b` if the parent of the block is open, else `0b` if it is closed.

The state of a treetable is contained in the path table `P`. For example, here is the state `P4` of the table `R4`:

```
n                       | v
------------------------| -
(`symbol$())!`symbol$()| 1
(,`A)!,`a               | 1
`A`B!`a`f               | 1
`A`B`C!`a`f`n           | 1
```

The visible function takes a path table and returns only those instructions which compute blocks which lie along visible paths:

```
q)visible P4
(`symbol$())!`symbol$()
(,`A)!,`a
`A`B!`a`f
`A`B`C!`a`f`n
```

Let's simulate closing `R4` at `A=`a`. The result `R5` should match `R1`, the initial, minimal treetable:

```
q)P5:closeat[P4;G;`a]
q)P5
n                       | v
------------------------| -
(`symbol$())!`symbol$()| 1
(,`A)!,`a               | 0      <- closed at A=`a
`A`B!`a`f               | 1
`A`B`C!`a`f`n           | 1

q)visible P5
(`symbol$())!`symbol$()

q)R5:construct[T;G;P5;A]
q)R5~R1
1b
```

Now we'll reopen R5 at A=`a. P6 should match P4 and the resulting treetable R6 should match R4:

```
q)P6:openat[P5;G;`a]
q)P6~P4
1b
q)R6:construct[T;G;P6;A]
q)R6~R4
1b
```

`openat` and `closeat` are projections of the underlying function `at`:

```
at:{[b;p;g;n]p,([n:enlist(count[n]#g)!n,()]v:enlist b)}

openat:at 1b
closeat:at 0b
```

`at` relies on the fact that catenation to a dictionary is *upsert*: append if the key is new, else update. For example:

```
q)d:`a`b`c!10 20 30
q)d,`c`d!40 50
a| 10
b| 20
c| 40
d| 50
```

`at` is trivial: flip the visibility bit for an instruction in the path table. The heavy lifting is performed by the `visible` function:

```
visible:{[p]
 q:parent exec n from p;
 k:(reverse q scan)each til count q;
 n where all each(exec v from p)k}
```

The first line computes the parent-vector `q` from the key of the path table `p` (see section 2 above). Line 2 computes the list of paths from the root to all nodes. Line 3 performs a running logical-and scan (q keyword: `all` down the boolean states of each path.

Here is a transcript of the console for an example run:

```
q)P5
n                      | v
-----------------------| -
(`symbol$())!`symbol$()| 1
(,`A)!,`a              | 0
`A`B!`a`f              | 1
`A`B`C!`a`f`n          | 1

q)p:P5

q)q:parent exec n from p
q)q
0 0 1 2

q)k:(reverse q scan)each til count q
q)k
,0
0 1
0 1 2
0 1 2 3

q)exec v from p
1011b

q)(exec v from p)k
,1b
10b
101b
1011b

q)all each(exec v from p)k
1000b

q)where all each(exec v from p)k
,0

q)n where all each(exec v from p)k
(`symbol$())!`symbol$()
```

There are good reasons for breaking out the state in this way.

In our example, the underlying table `T` has 1000 records, and the treetable in its fully opened state, in which all leaves of `T` and all aggregations are constructed, has 1206 records. The state-table therefore has 206 instructions, each of which corresponds to a complete scan of `T`. (See `Q2` and `S2` in [1].) Clearly, this can get expensive. For example, where the underlying table contains millions of records and hundreds of aggregated columns, and the tree-structure is deep and bushy. Moreover, we cannot rule out the possibility that the underlying table is the target of frequent updates; for example, if it is connected to a real-time data-source. In that case, we cannot even be confident that the structure of the tree won't change. (See the `valid` function in [1].)

For these reasons, the design is deliberately *lazy*: we compute only as much of the tree as the path-table directs.

6. Sort

The APL sorting primitives *grade-up* and *grade-down* appear in q as the keywords `iasc` and `idesc`.

We can use `over` to do multi-column sorts:

```
msort:{x y z x}
```

`msort` is a function of three arguments: `x`, an index vector; `y`, a permutation function; and `z`, a list. Thus: `x` permuted by the result of applying `y` to `z` permuted by `x`.

Let `v` be a list of two vectors:

```
q)v
0 2 4 4 3 0 4 3 0 3
0 3 1 4 1 3 1 3 1 2
```

Sort `v 0` descending within `v 1` ascending:

```
q)i:msort/[til count first v;(idesc;iasc);v]
q)i
0 2 6 4 8 9 7 1 5 3

q)v@\:i
0 4 4 3 0 3 3 2 0 4
0 1 1 1 1 2 3 3 3 4
```

`msort/` repeatedly permutes `x` by the result of applying `y` to `z` permuted by `x`.

`msort` can be used to sort dictionaries of vectors:

```
q)d:`a`b!v
q)d
a| 0 2 4 4 3 0 4 3 0 3
b| 0 3 1 4 1 3 1 3 1 2

q)d@\:msort/[til count first d;(idesc;iasc);d]
a| 0 4 4 3 0 3 3 2 0 4
b| 0 1 1 1 1 2 3 3 3 4
```

and tables:

```
q)t:flip d
q)t
a b
---
0 0
2 3
4 1
4 4
3 1
0 3
4 1
3 3
0 1
3 2
```

```
q)t msort/[til count t;(idesc;iasc);flip t]
a b
---
0 0
4 1
4 1
3 1
0 1
3 2
3 3
2 3
0 3
4 4
```

Our problem is to adapt `msort` to apply recursively to the hierarchically-related blocks of a treetable.

In our example, `R4` has 25 records and blocks at four levels:

```
n_         | A B C counts v      w
-----------| --------------------
`symbol$()|       1000   479131
,`a        | a     224    106670
`a`f       | a f   28     13952
`a`f`n     | a f n 7      2867   x
`a`f`n`0   | a f n 908    908    x
`a`f`n`1   | a f n 256    256    x
`a`f`n`2   | a f n 401    401    x
`a`f`n`3   | a f n 288    288    x
`a`f`n`4   | a f n 543    543    x
`a`f`n`5   | a f n 258    258    x
`a`f`n`6   | a f n 213    213    x
`a`f`o     | a f o 7      3707   y
`a`f`p     | a f p 7      3640   z
`a`f`q     | a f q 7      3738   w
`a`g       | a g   28     14948
`a`h       | a h   28     12190
`a`i       | a i   28     13535
`a`j       | a j   28     13835
`a`k       | a k   28     12945
`a`l       | a l   28     13643
`a`m       | a m   28     11622
,`b        | b     200    100048
,`c        | c     192    90541
,`d        | d     192    92853
,`e        | e     192    89019
```

`R4` is implicitly hierarchical. The typical approach to operating on such structures is to apply a 'flat' algorithm like `msort` recursively. The deprecated function `rsort` in [1] exemplifies this approach.

But there is a better way. We want an 'array solution' where the iteration is handled covertly by primitives. And we have one. Our solution will (i) convert the parent-vector into a list of child-vectors; (ii) use the child-list to partition the treetable into child-blocks; (iii) sort each child-block; (iv) use the key of the treetable to reassemble the sorted blocks into a treetable.

Let's step through it using `R4`.

First, de-key `R4`:

```
q)t:0!R4
```

Next, compute the parent-vector of `t` from column `n_`, the path-list:

```
q)parent:{[n]n?-1_'n}

q)n:exec n_ from t
q)p:parent n
q)p
0 0 1 2 3 3 3 3 3 3 2 2 2 1 1 1 1 1 1 1 0 0 0 0
```

Next, compute the child-list from `p`:

```
q)children:{[p]@[(2+max p)#enlist();first[p],1+1_p;,;til count p]}

q)i:children p
q)i
,0
1 21 22 23 24
2 14 15 16 17 18 19 20
3 11 12 13
4 5 6 7 8 9 10
```

`i` is a list of indices into `t` such that `t i` is a list of the subtable blocks of `t`:

```
q)t i
+`n_`A`B`C`counts`v`w!(,`symbol$();,`;,`;,`;,1000;,479131;,`)
+`n_`A`B`C`counts`v`w!((,`a;,`b;,`c;,`d;,`e);`a`b`c`d`e;`````;`````;224 200 1..
+`n_`A`B`C`counts`v`w!((`a`f;`a`g;`a`h;`a`i;`a`j;`a`k;`a`l;`a`m);`a`a`a`a`a`a..
+`n_`A`B`C`counts`v`w!((`a`f`n;`a`f`o;`a`f`p;`a`f`q);`a`a`a`a;`f`f`f`f;`n`o`p..
+`n_`A`B`C`counts`v`w!((`a`f`n`0;`a`f`n`1;`a`f`n`2;`a`f`n`3;`a`f`n`4;`a`f`n`5..
```

For example, block 1 is:

```
q)t i 1
n_ A B C counts v      w
------------------------
a  a     224    106670
b  b     200    100048
c  c     192    90541
d  d     192    92853
e  e     192    89019
```

Suppose we have a single sorting operation `o` and a single column `c`:

```
q)c:enlist`v
q)o:enlist iasc
```

Sort each block:

```
q)j:msort[t;c;o]each i
q)j
,0
24 22 23 21 1
20 15 18 16 19 17 2 14
3 12 11 13
10 5 9 7 6 8 4
```

We now have the permutations we need to sort each block of `t`:

```
q)t j 1
n_ A B C counts v      w
------------------------
e  e     192    89019
c  c     192    90541
d  d     192    92853
b  b     200    100048
a  a     224    106670
```

Our adaptation of `msort` for treetables is:

```
msort:{[t;c;o;i]i{x y z x}/[til count i;o;flip[t i]c]}
```

As the last step, we need to mesh the permutations to give us the single permutation vector `v` /such that `t v` is `t` in sorted order.

To do that, we first compute the reordered keys of the blocks:

```
q)m:n j
q)m
,`symbol$()
(,`e;,`c;,`d;,`b;,`a)
(`a`m;`a`h;`a`k;`a`i;`a`l;`a`j;`a`f;`a`g)
(`a`f`n;`a`f`p;`a`f`o;`a`f`q)
(`a`f`n`6;`a`f`n`1;`a`f`n`5;`a`f`n`3;`a`f`n`2;`a`f`n`4;`a`f`n`0)
```

Then, to mesh the keys we insert each path-list into the appropriate slot of the mesh of the previous path-lists. This is our function `pmesh`:

```
pmesh:{i:1+x?-1_first y;(i#x),y,i _ x;()}
```

We apply it over `m` to give us the permuted path-list of the sorted table:

```
q)k:pmesh over m
q)k
`symbol$()
,`e
,`c
,`d
,`b
,`a
`a`m
`a`h
`a`k
`a`i
`a`l
`a`j
`a`f
`a`f`n
`a`f`n`6
`a`f`n`1
`a`f`n`5
`a`f`n`3
`a`f`n`2
`a`f`n`4
`a`f`n`0
`a`f`p
`a`f`o
`a`f`q
`a`g
```

Finally, we look up `k` in `n`, which gives us the index-vector `v` which permutes `n` into `k`:

```
q)v:n?k
q)v
0 24 22 23 21 1 20 15 18 16 19 17 2 3 10 5 9 7 6 8 4 12 11 13 14
```

and hence `t` into `t` upsorted by `v`:

```
q)t v
n_          A B C counts v      w
--------------------------------
`symbol$()        1000   479131
,`e         e     192    89019
,`c         c     192    90541
,`d         d     192    92853
,`b         b     200    100048
,`a         a     224    106670
`a`m        a m   28     11622
`a`h        a h   28     12190
`a`k        a k   28     12945
`a`i        a i   28     13535
`a`l        a l   28     13643
`a`j        a j   28     13835
`a`f        a f   28     13952
`a`f`n      a f n 7      2867   x
`a`f`n`6    a f n 213    213    x
`a`f`n`1    a f n 256    256    x
`a`f`n`5    a f n 258    258    x
`a`f`n`3    a f n 288    288    x
`a`f`n`2    a f n 401    401    x
`a`f`n`4    a f n 543    543    x
`a`f`n`0    a f n 908    908    x
`a`f`p      a f p 7      3640   z
`a`f`o      a f o 7      3707   y
`a`f`q      a f q 7      3738   w
`a`g        a g   28     14948
```

Assembling the steps:

```
tsort:{[t;c;o]
 n:exec n_ from t;
 i:children[parent n]except enlist();
 j:msort[0!t;c;o]i;
 n?pmesh over n j}
```

7. Conclusion

q is a language of lists and dictionaries. By adding tables (lists of dictionaries) and keytables (dictionaries of tables), q inverts the traditional relationship between database and programming language.

In the familiar model, tables live in a database. Programs extract data from tables in the database, and insert data into them. Other programs, usually written in some special database-y language, can be attached to database tables as 'triggers' If you're used to this sort of thing it doesn't seem so onerous. If you're not, it feels like sorting rice-grains while wearing mittens.

In q, tables and keytables are first-class entities whose parts are first-class. You assign them, transform them, bust them apart, stick them in lists, and pass them into and out of functions, just the way you do with lists and dictionaries. And that's because they *are* lists and dictionaries.

In most applications, the built-in SQL-like syntax of q is perfectly adequate:

```
select/exec/update/delete … by … from … where …
```

But as the treetable example shows, it may be necessary to drop down to the functional level where the SQL keywords give way to the primitives `?` and `!` and the content of the queries is carried as q-object arguments to those primitives.

Acknowledgements.

Thanks to Attila Vrabecz for corrections and several black-belt one-liners.

References

1. http://www.nsl.com/q/treetable.q