Professor Dennis Shasha
2011

# q for smarties 3

https://code.kx.com/trac/wiki/QforMortals/tables
https://code.kx.com/trac/browser/kx/kdb+/sp.q

Sweet spots:

- Many columns where ordering (e.g. by time) is needed and operations on ordered data are desirable.
- Historical data where need only a few columns per query but table has many columns.
- Transaction processing (over 10,000 per second read-write transactions) on data that can fit into memory.

Sour spot: Transaction processing on data that cannot fit into memory (take some work).

**0. Run setup** setup.q

```
stock: `ibm`bac`usb`bac`bac`usb;
stock,: `ibm`bac`usb`ibm`bac`usb;
stock,: `ibm`ibm`usb;
price: 1.1 2.4 3.3 3.0 2.8 1.8;
price,: 3.1 4.4 1.3 2.0 3.8 5.8;
price,: 5.1 4.3 3.3;
amount: 100 200 300 400 500 300;
amount,: 300 200 400 600 700 800;
amount,: 700 800 700;
time: 09:03:06.000 09:03:23.000 09:04:01.000 09:05:01.000 ↩
   09:06:01.000 09:07:01.000;
time,: 10:03:06.000 10:03:23.000 10:04:01.000 10:05:01.000 ↩
   10:06:01.000 10:07:01.000;
time,: 11:03:06.000 11:03:23.000 11:04:01.000;
trade:([]stock; price; amt:amount; time);
save `:trade.csv; / to write a csv file
```

Things to note:

1. Insert into a table by creating lists and then assigning.
2. The initial brackets are for key fields. There are none here.
3. Saving to a csv file is possible for small files (up to a few hundred megabytes).

Find all the tables by

```
tables `
```

Task: Import a table of stock prices in csv format (‘trade.csv as written above) and find the average price per stock and write that to a csv file

```
    q readavg.q
```

Reading in uses a type field (S for strings, F for floats, and T for times) with a delimiter of comma.

```
mytrade: ("SFFT"; enlist ",") 0: `trade.csv
```

If you know SQL, then kdb will be easier to learn. It is a semantic extension (arrable model, as I will explain) but has somewhat different syntax. Let's look at the syntactic differences first.

```
myavg: select avg price by stock from mytrade
```

update columns

In standard SQL the "group by" goes after the "from" and "where" clauses. Here the "by" clause goes before "from".

```
save `:myavg.csv
```

Exercise 1. (Middle) The volume weighted average price per trade is the average of the price times the volume. Compute the volume weighted average price per stock

```
        q vwap.q
```

Can create new columns by update, e.g.

```
mytrade: update weightedprice: price*amt from mytrade
```

The very powerful semantic extension that kdb offers is that it is based on ordered columns and therefore it can support queries that make use of that order. You can therefore create functions that work on arrays and run them. Or you can use built-in ones when available.

```
/ given a list of prices, get the 2 way moving average

/ moving average version -- roll your own
movavg2:{[x] (x[0]),avg each ((-1) _ x),'(1 _ x)}
mytrade: ("SFFT"; enlist ",") 0: `trade.csv
myavg2: select movavg2 price by stock from mytrade

/ collective average version
mytrade: ("SFFT"; enlist ",") 0: `trade.csv
myavg2: select avgs price by stock from mytrade
save `:myavg2
```

Note the different method of saving. It doesn't work to save it to a csv file because the output is not in first normal form.

Also note that this may not be meaningful unless you know that mytrade is ordered by time. This is easily corrected in both the roll-your-own version

```
myavg2: select movavg2 price by stock from (`time xasc mytrade)
```

and using the built-in moving average

```
myavg2: select mavg[2;price] by stock from (`time xasc mytrade)
```

> Exercise 2. Compute the 4 moving average per stock using mavg and optionally in a roll-your-own way.

Hints for the roll-your-own crowd: Take care of the first three moving averages using the avgs function and then go on from there using the difference of sums. Display the result and save it as a q file. If possible, write a generic moving average function.

```
    q readmovavglong.q
```

Note: You can load the table in another q file by saying

```
load `:myavg2
```

Let's say we want to generate a random table having 100,000 rows from the stocks ‘ibm ‘hp ‘amaz ‘goog ‘aapl, prices in the range 20 to 400, and volumes in the range 100 to 100000 but multiples of 100, times arbitrary in the range from 10 AM to 4 PM.

```
    q gendata.q

n: 100000;
stocks: `ibm`hp`amaz`goog`aapl;
rantrade:([]stock: n?stocks; price: 20 + n?380.0;amount: 100*(1+n?1000);
  time: 10:00:00.000 + n?06:00:00.000);
```

Points to note:

1. n ? stocks
2. 20 + n ? 380.0
3. n ? 06:00:00.000

```
save `:rantrade.csv
```

Use the table generated and select the name and last price of each stock into a new table pricecol.

```
endofdayprices: select last price by stock from (`time xasc rantrade)
```

Notes:

1. You do not have to put "stock" in the select clause since already in the by. In fact, you shouldn't do it.
2. Because the columns aren't ordered in the original rantrade, this does not give us the last stock by time. That is why we use xasc.

Exercise 3. (Easy) Find the average value of each trade where time > 12:00:00

```
    q volweight.q
```

A few notes on inserting data.

```
/ get the prices column as a list.
/ price is the second column, but col numbers start at 0
(value flip rantrade)[1]

`foo insert rantrade; / note that insert creates a table if it doesn't exist

count foo

`foo insert rantrade / or if it already exists

count foo
```

Another way to insert data. Can also insert into a table using columns:

```
newstocks: n?stocks;
newprice: 20 + n?380.0;
newamount: amount: 100*(1+n?1000);
newtime: 10:00:00.000 + n?06:00:00.000;

`rantrade  insert (newstocks; newprice; newamount; newtime);
```

Exercise 4. (Easy) Generate a second table having the same schema as in the random table and bulk insert it into the random table. Then form columns corresponding to this schema, each of the same length as the columns of the original random table. Bulk insert those columns. Which is faster? Table insert or column insert? (Use before the statement).

```
    q bulkinsert.q
```

Tables can be sorted based on different fields. e.g. let's sort the table rantrade based on price.

```
ranprice: `price xasc rantrade

x: exec price by stock from rantrade
x
```

This gives a dictionary that has stock-price pairs. Because stock names are keys, you can ask for their values.

```
x[`aapl]
```

Exercise 5. (Optional, Hard) Compute the correlation of the prices of every pair of stocks in time order of their trades. Use the trade.csv table. Each stock has the same number of values in the trade table, but they are not necessarily in time order. You will have to bring stock-price pairs out and then write a function to compute over them.

```
q findcorr.q
```

Excercise 5h. (Optional, Slightly Harder variant) Generate a random stock-trade table and then compute correlations between every pair of stocks. If they don't have the same length then truncate to the smaller size. Ensure the two vectors are ordered in time.

```
q findcorrharder.q
```

Sometimes, one may want to declare a key for a table. Suppose that each of these companies is associated with a unique state which constains its headquarters.

```
stocks: `ibm`hp`amaz`goog`aapl;
states: `ny`ca`wa`ca`ca;
```

Then declare a key for the table as follows.

```
stock:([mystock: stocks] place: states);
```

Note that the first field is in brackets. That is a signal to show that no two rows should have the same mystock value. Having these keys enables us to do table joins in an easier way. (You can undo the keyness of the table by typing 0!`stock.) First, though, note the new way that the stock field in rantrade is specified. The construct `stock means that all elements here should be subsets of the key of the stock table (the column entitled mystock).

```
n: 100000;
rantrade:([]stock: `stock$n?stocks; price: 20 + n?380.0;
  amount: 100*(1+n?1000); time: 10:00:00.000 + n?06:00:00.000);
    / first field must be called stock and must reference the key
    / field of the stock table. That field can be called something ↩
       else.

select stock, stock.place from rantrade
```

Note that the field referencing the table stock must be called stock. This permits an implicit join from rantrade to stock through the field stock in rantrade. The field stock in rantrade refers to the foreign key mystock in stock. We know this because in the rantrade schema we describe the field stock as

```
`stock$n?stocks.
```

This says that the stock field of rantrade is a subset of mystock in stock and allows a row in rantrade to be linked to a single row in table stock. (Many rows in rantrade may be linked to the same row in table stock.)

Exercise 6. (Easy) Add an address table that links the stocks `ibm`hp`amaz`goog`aapl with their state addresses and then find the stock-address pairs of all stocks whose average price is above 80.

```
q findgoodaddress.q
```

Exercise 7. (Harder) Change the name of stock to johnstock in rantrade and do so consistently. Also create a second level of hierarchy so that ca is warm, ny is cold, and wa is tepid. Then use that.

```
q findgoodaddressvar.q
```

A little interlude on debugging:

Sometimes the errors messages can be challenging to figure out. For example, if you declare rantrade as we did originally:

```
stocks: `ibm`hp`amaz`goog`aapl;
states: `ny`ca`wa`ca`ca;

stock:([mystock: stocks] place: states);


/ declare rantrade as before
rantrade:([]stock:n?stocks;price:20+n?380.0;amount:100*(1+n?1000);
  time:10:00:00.000+n?06:00:00.000);

/ Then try the statement
select stock, stock.place from rantrade
```

This will encounter an error:

```
k){0N!x y}
'type
@
"q"
"select stock, stock.place from rantrade"
```

To debug these, type y and you will tend to see the problem statement. In this case the definition of rantrade fails to create an association between rantrade and the stock table.

Interlude: If you want more exercise, go to
https://code.kx.com/trac/wiki/QforMortals2/queries_q_sql#Parameterized-Queries
and start at section entitled Examples (section 14).

Interlude: eval and run Sometimes it is useful to assemble an SQL string and then just execute it. One can do that in kdb. For example, put these statements in a file that has rantrade defined and execute:

```
value "select stock, price from rantrade"
```

One can assemble such queries more formally by forming such strings on the fly.

Exercise 7. (Medium) Write a function that take an arbitrary table, an arbitrary target column expression, and an arbitrary where clause and can apply to any table having those columns and for which the where clause is appropriate. Test it on trade.csv.

```
q arbquery.q
```

Spreading Load:

When queries are expensive, it's useful to spread them around on different servers. The idea here is that we'll have a generic server that will receive requests from application-specific clients and send them to application-specific slaves.

Please look at https://code.kx.com/trac/wiki/Cookbook/LoadBalancing

Here might be a client:

```
h: hopen `:localhost:5001

neg[h]"select avg price*amount by stock from rantrade"; h[]
neg[h]"select max price*amount by stock from rantrade"; h[]
neg[h]"select min price*amount by stock from rantrade"; h[]
neg[h]"select var price*amount by stock from rantrade"; h[]


Here might be a slave server:

n: 100000;
stocks: `ibm`hp`amaz`goog`aapl;
rantrade:([]stock:n?stocks;price:20+n?380.0;amount:100*(1+n?1000);
  time:10:00:00.000+n?06:00:00.000);
```

The generic router is mserve.q, copied from the website and that we will treat as a black box.

Exercise 8. (Middle) Communicate a set of read only requests to a master asynchronously and then they are routed to slaves. Import into each slave the trade.csv table and then have the client send several requests, each one about a single stock.

```
        q mserve.q -p 5001 3 slaveserver.q
    q masterclient.q
```

That's all you do. No need to start slaveserver.q. mserve.q does this for you. Note that the client issues async calls and then waits for the response with h[].

For future reference... It is sometimes useful to access a website, scrape the result, and present the result as a table. https://code.kx.com/svn/cookbook_code/yahoo.q presents a nice example. Let's analyze it. I've added in some print statements that begin with 0N!

Even more useful is a statement that creates a deliberate type error

```
1 + `x;
```

This causes an error, but from this error, one can inspect variables and even change them. Then you resume by typing

```
:
```

Exercise 9. (Debugging and modification) Web access. Take yahoo.q and put in a deliberate type error and then modify enddate. Modify the result to return the Sym,Date,Close columns.

```
q yahoomod.q
```