AquaQ Training

Kdb+ Training Exercises

email:
support@aquaq.co.uk

web:
www.aquaq.co.uk

AquaQ Analytics

Revision History

Revision	Date	Author(s)	Description
0.1	February 3, 2014	AquaQ	First version created

Contents

1 Atoms and Basic Operations **4**

2 Lists **6**

3 Dictionaries **9**

4 Table Operations **11**

5 Functions **13**

6 Scripts **15**
6.1 Exercises from Notes 15
6.2 Easy 16

7 Tables and Queries **17**
7.1 Tables 17
7.2 Queries 18
7.3 Easy 19
7.4 Moderate 20
7.5 Hard 21

8 Joins **22**

9 Adverbs **23**

10 Basic IPC **25**

11 Web Access **27**
11.1 Exercises from Notes 27
11.2 Easy 28
11.3 Moderate 29

12 Tables on Disk **30**

13 ETL **32**

14 Inter Process Communication **33**
14.1 Connections . . . 33
14.2 Synchronous Calls . . . 34
14.3 Asynchronous Calls . . . 35

15 HDB Queries **38**
15.1 Selects . . . 38
15.2 Aggregations . . . 39
15.3 Calculated Data . . . 41
15.4 Joins . . . 43

16 Tickerplant **45**
16.1 Tickerplant Setup . . . 45
16.2 Feedhandler . . . 46
16.3 RDB and HDB Rollovers . . . 47
16.4 Real Time Analytics . . . 48
16.5 Daemonizing . . . 50

17 File Loading **51**
17.1 Basic Loading . . . 51
17.2 Loading Big Data Sets . . . 52

18 Creating a Trade Report **54**
18.1 Report specification . . . 54
18.2 Tasks . . . 54
18.3 Optional Extras . . . 56

Chapter 1

Atoms and Basic Operations

1. Start a new q session.
2. Define the following variables:
   - $a = 5$
   - $b = a - 3$
   - $c = 3b + 1$
3. Repeat question 2 in one single assignment statement.
4. Calculate $\frac{a^2-3c}{b}$ and store the answer as the variable y. (Use as few brackets as possible)
   - What is the type of y?
   - What is the result of typing `y=b` and `y~b` in the q session? Explain the difference.
   - How can you amend y so that `y=b` and `y~b` agree?
   - Multiply the variable b by 5 in place.
5. Find the type values of the following objects:

```
123
123f
2014.09m
`me
"you"
```

6. Which of the following objects have the same type?

```
4.29999871
7j
4.2e
0b
9.8
```

What are the types of the other objects?

7. Get today's date (.z.d) and store it as the variable *d*.
   - Look at the contents of *d*
   - Calculate the number of days since last Christmas.
   - Find out on which day of the week that the 10th of January 2011 was (hint: try using mod with a date)
8. Define the following items, and then cast as appropriate to the stated type:
   - `d1:"2014.01.01"` to a date
   - `d2:`2013.12.10` to a date
   - `n1:3.14` to an integer
   - `n2:"2"` to an integer
   - `a1:"abcde"` to a symbol
   - `a2:"abcde"` to the ascii codes of the letters.
9. Show all of the variables defined in the current q session.
10. Close the current session.

Chapter 2

Lists

1. Create two lists, a with values 1 2 3 and b with values 4 5.
   - Make one new list c from a and b.
   - Sum the elements of c
   - Multiply all of the elements in c by 10.
2. Define a new empty list, d.
3. Redefine d to be an empty list of type integer.
4. Add 5 random elements to d.
5. Create a list e with one element.
6. Create a list, $list1$, with 20 random values between 3 until 30.
   - Find the maximum, minimum and average value of $list1$.
   - Find the number at index 10 in $list1$.
   - Find the 20th number in $list1$.
   - Are any of the following numbers in the list?

     ```
     3 5 7 11 13 17
     ```

   - Multiply each element of $list1$ by 3.
   - Add to each element of $list1$ its index in the list, i.e. add 0 to the element at index 0, add 1 to the element at index 1 and so on.
   - Find all the even numbers in the list; how many are there?
   - Take the first 10 items of $list1$ and cast them to dates; why do you get these dates?
7. Create the matrix m:

```
1 2 3
4 5 6
7 8 9
```

- Get the middle column of m and save it as a new variable a
- Replace the middle row of m with a
- Transpose m and store as mm
- Join an extra row to mm, consisting all of 10s
- Join another row, 11 12 13

8. Create the following nested list:

```
1 2 3
`a`c`b
10 11 12 14f
100011b
```

- Find the type of each of the rows.
- Collapse the nesting of this list.

9. Define the strings $s1$:"Hello" and $s2$:"World".

- Join them together to form a new string s:"Hello World".
- Find the index of the letter "W" in s.
- Find the index of the last "l" in s.
- From the string s, remove "Hello" from the start and add "of Warcraft" on the end. Try to do this in one single operation.

10. The following data regards the usage of some cool computer games from just a few years ago by our office team.

Game Title	Platform	Level of User
Crash Bandicoot	PS1	7 9 6 8 4 1
Streets of Rage	Sega Megadrive	2 1 3 5
Echo the Dolphin	Sega Megadrive	8 5 7 5 4 4
Crash Bandicoot 2	PS1	10 2 1
Sonic and Knuckles	Sega Megadrive	4 2 1 10
Micro Machines	PS1	8 4 9 5 8 10
Pokemon Red	Gameboy	0 3 2 10 8 5 6 10
Super Mario Brothers	Gameboy	6 0
Mega Bomberman	Sega Megadrive	8 4 8 9 5 1 7
Zelda	Gameboy	0 10 9 5 2

- Create three lists, one for each column of data in the table. The lists *games* and *platform* should be of type symbol (abbreviate as you wish!), with *levels* a list of nested lists, of type int
- Create another list, which is the number of users for each game
- Calculate the average user level for each game
- Create a boolean list inidicating where the average user level is more than 6
- For which games is the average user level more than 6?
- Sum the users for each platform. Which was the most popular?
- Remove from all of the lists the data for the gameboy.

Chapter 3

Dictionaries

1. Create a dictionary with the letters from a to z, with their corresponding numerical value, i.e:

```
a| 1
b| 2
c| 3
...
```

   - By indexing into the dictionary, find the numerical value of your name.
   - A list of letters are held under .Q.a for small case and .Q.A for uppercase. Rename the keys of your dictionary to be the uppercase letters.

2. Using another dictionary of your own design, change the uppercase words "HELLO", "WORLD" to lowercase.

3. Create a dictionary morse, which contains the letters of the alphabet from n through to s as the keys and the morse code for those letters as the values. (Indicate the values as you like; maybe 1s and 0s?)

   - We forgot that the numbers from 0 to 4 are useful too; join these on to the existing dictionary.
   - Form the phrase SOS in morse code.

4. That was quite a topical question for Belfast. Here's another one, about the weather. Create two dictionaries, which will contain the following data with the places as keys, and one dictionary each for each year:

```
Places: `London`Edinbugh`Belfast`Manchester`Tobermory`Portsmouth`Cardiff
Rainfall total (2013): 557, 704 ,944, 867, 1681, 674, 1152
Rainfall total (2012): 600, 854, 1020, 955, 1789, 544,
```

   - Extract the keys from the first dictionary

- Extract the values from the first dictionary
- Create a new dictionary with the sum of the annual rainfall for both years.
- Find the average rainfall over the two years for each city
- For which locations was the annual rainfall more in 2013 than 2012?
- Find the average rainfall over the two years for the UK

5. Create a dictionary containing all the variables in the workspace as the key, and their values as the values of the dictionary.

Chapter 4

Table Operations

1. Create three lists, a, b, and c, each with 4 elements.
   - Make a new table, t with each column consisting of the lists a, b, and c.
   - Make a dictionary d from the table t.
2. Create an empty *trade* table, with 4 columns:
   - sym (symbol type)
   - side (char)
   - size (int)
   - price (float)
3. Create another empty table, *lasttrade*, which is a copy of *trade*. Key this by sym.
4. Is there a difference in the metadata between *trade* and *lasttrade*? What about the type? What do each of the columns in meta represent?
5. Using three join commands add a row each time containing data from the following to *trade*:
   - sym taken from IBM, MSFT, AAPL
   - side taken from “B" (buy) and “S" (sell)
   - sensible/consistent values for other columns
6. Using a single join operation add 3 more made-up rows into *trade*, with the same conditions on the data.
7. Fill the table *lasttrade* with the data from *trade*.
8. The manager at a local shop wants to create a database of orders that he can view and get information from easily.

- Create the table *stock*. It should contain the following data, in appropriate format:

```
q)stock
item      brand       priceperunit order
---------------------------------------
sodabread frysareus   1.5          50
bacon     prokstop    1.99         82
mushrooms veggiestuff 0.88         45
eggs      veggiestuff 1.55         92
```

- The manager forgot to add in tomatoes. He must include an order of 70 units at 1.35 each, from Veggiestuff. Add this permanently to the table.
- View the meta data for the table
- Key the table according to item and brand
- How does the meta data compare now to the previous version, unkeyed?

9. The manager decides to do his bulk orders together with a food stall trader at the market. They hope to save money that way.

- The following table is the information from the trader for his weekly orders.

```
item      brand       priceperunit order
---------------------------------------
sodabread frysareus   1.5          200
bacon     prokstop    1.99         180
mushrooms veggiestuff 0.88         110
eggs      veggiestuff 1.55         210
tomatoes  veggiestuff 1.35         100
```

- Create this table, *trader*, with the appropriate format
- Key the table by item and brand
- Create a new table, *totalorders*, which has the sum of both orders from the traders - drop the *PricePerUnit* column from each table first
- They can both reduce the prices to be 75% of the original price by ordering together. Create a list *newprices*.
- Join this on as a new column to *totalorders*
- How much money per week will the manager of the local store save by working with the trader? What about the trader?

Chapter 5

Functions

1. Write a monadic function, **f**, which calculates the square of a number (using an explicitly defined parameter).
2. Execute **f**, with argument a, and assign the result to new variable b.
3. Write a dyadic function, **g**, which calculates the square of the first argument, divided by the square of the second argument (using implicit parameters).
4. Execute **g**, with arguments a and b. Store the result of this function in a variable called c.
5. Create a dyadic function **f1** which indicates whether the product of two numbers is greater than their sum.
6. Let $a = 5$, $b = a - 5$ and define $c = \frac{b}{a} + \frac{a}{b}$ using only one set of brackets.
7. Define a function **g1**, which calculates the polynomial $x^5 - 3x^2 + 5$ and determine the value of the polynomial when $x = 4, x = 8$ and $x = 6.7$.
8. Calculate the following without using brackets: $-5\left(\frac{6}{3-1}\right) + 7$.
9. Create a function that calculates the area of a triangle. Find the area of a triangle of height 7cm and base 10cm.
10. Create a function that calculates the sum of two squares of a number x. Find the sum of two squares of 18.
11. Create a function that takes three values a, b and c and calculates $a^3 + b^2 + c$. Find the value of the function at 13,3,6.
12. A rough calculation of the BMI of a person takes their weight in kilograms and divides it by their height in metres squared. Create a formula which will calculate the BMI for any individual, without any explicit parameters.

13. Degrees Celcius is converted to degrees fahrenheit by multiplying by $\frac{9}{5}$ and adding 32, create a formula for this using only implicit parameters.

14. Using a parameter called perimeter, create a formula called area which will take the perimeter of a square and find it's area. This should only require one set of round brackets.

15. **xexp** is a built-in dyadic function which calculates powers, for example 2 **xexp** 3 gives 8. Use this to re-create the formula in the previous question without explicit parameters.

16. A car depreciates at a rate of 15% for the first three and 8% for the next three years. Create a function k to find the value of a car after 6 years. Find the value of a car of starting value 15000GBP after 6 years.

17. A grocer wants to write a function that he can use each day to find out how much his fruit orders will cost. Using the following data create a function to calculate how much his orders will cost each day. Apples, oranges, bananas and pears cost \$1.59, \$1.99, \$0.99 and \$2.49 per kilo respectively and the grocer has the following orders:

Fruit	Day 1	Day 2	Day 3
apple	2	4	4
orange	2	5	7
banana	1	1	4
pears	0.5	2	0.5

Chapter 6

Scripts

6.1 Exercises from Notes

1. Write a script *inner.q* to:
   (a) Define a niladic function, **prnt**, to print *Loading inner* to the screen.
   (b) Define a monadic function, **recip**, to return the reciprocal of a number.
   (c) Define a triadic function, **average3**, to return the average of 3 numbers.

   and save this script to the kdb+ installation directory.
2. Write another script *outer.q* which will:
   (a) Print out *Loading outer* to the screen.
   (b) Load script *inner.q*
   (c) Run **prnt**.

   and save this script to the kdb+ installation directory.
3. Start a q process, loading outer.q from the command line and:
   (a) Execute **average3** with arguments 3, 5, 10
   (b) Execute **recip** with argument 3

6.2 Easy

1. Create a script called *orders* which generates the complete keyed table *stock* and the table *trader* from Exercises 4.
2. Create a script called *HR* which generates the tables which can be seen in Exercises 7.1.
3. Use the ? function[1] to generate an *employee* script containing the *employee* table. This includes employee id (primary key) and randomly generated marital statuses, city keys, salary band keys, genders and hire dates (before 2011). Edit the HR script to load this new employee's script.
4. Load *shop.q* into kdb+ and view stock table.
5. Load *HR.q* into kdb+ and view country and employee tables.
6. View the tables in the current q session.
7. Edit the previous scripts so that they show the names of all the tables available when they are loaded.

[1] Hint: The random function (?) can be used to select a defined number of elements from a list. Generally, one can choose x elements from L with the following syntax:

$$x?L$$

For example;
To choose 5 integers between 0 and 10 inclusive, use 5?11
To choose 20 floats between 0 and 10 inclusive, use 20?11f
To choose 3 symbols from the first five letters of the alphabet use 3?`$a$`b`$c$`d`e
To choose 300000 characters from M and F use 300000?"MF"

Chapter 7

Tables and Queries

7.1 Tables

The law firm *Legalfirst* needs to store their HR data for later use. They want separate tables for different information and will connect them together at a later stage, to a larger table not contained on a platform that is instantly accessible to kdb+.

1. Create the following table:

```
q)country
id city       country
---------------------
1  london     uk
2  manchester uk
3  newyork    usa
4  toronto    canada
```

2. Create an empty table called bonus, with columns called year (of type integer) and bonusvalue (of type float).

3. Insert the following data into the table:

```
q)bonus
year bonusvalue
-----------
1990 2000
2000 1500
2010 1000
```

4. Create a keyed table for salary bands called 'bands' which contains the following data.

```
q)bands
code| salary
----| -------
A   | "15-20"
B   | "20-30"
C   | "30-40"
D   | "40+"
```

where A, B, C and D are symbols. Select an appropriate type for salary.

5. *Legalfirst* has decided to futher divide their salary ranges, change D to include salaries from 40k to 60 k and a new band E to include $60k+$.

7.2 Queries

1. Start a session, load 'fakedb.q' and create a database with 10000 quotes and 2000 trades.
2. Calculate the total volume traded for each sym and side, and store the result in a table 'aggr'.
3. Modify 'aggr' in place so the sell volumes are negative amounts.
4. Use the modified 'aggr' to calculate the final position, long or short, for each symbol.
5. Select all quotes for IBM, and store the result in a table 'ibmqt'.
6. Modify 'ibmqt' in place to add a new boolean column 'pricecross', indicating whether the bid price is greater than or equal to the ask price.
7. Using the modified 'ibmqt', find the number of quotes where the prices are crossed, and the number where they are not crossed.
8. Delete (in place) all trades that occurred after midday.

7.3 Easy

1. Load the script 'shop' and use kdb+ to find from 'stock' table:
   (a) the item with the largest order
   (b) all items which are supplied by *OrchardFresh*
   (c) the brand with the maximum price
   (d) the total number of orders
   (e) average order price according to brand
   (f) total order size for each brand.
2. Create a new column in the 'stock' table called 'total_order_price' which is the product of 'price_per_unit' and 'order'.
3. Create a new table called 'brands' which contains the columns 'brand' and 'total_brand_order_price'.
4. Load the fakedb.q script and create a trades and quotes table.
5. Return a table showing the ratio of trades to quotes for the syms in the input list and order this table from highest to lowest.
6. Return a table showing the maximum time between each trade and the maximum time between each quote for the input list of syms.
7. Find the sum bought and sum sold for each sym in the input list. Also add a field for the position assuming the side field is the client side of the deal.
8. Find the number of times the quote mid moved up, the number of times it moved down and the number of times it stayed the same. Do the same for the trades table with regards to price.
9. Find the number of trades executed inside the bid and the ask, the number of buys above the mid and the number of sells below the mid, by sym.
10. Reverse the order of the characters of each sym in the trade table.

7.4 Moderate

1. Load the school.q script included with the notes and view the tables present in the q session.
2. View the marks for class A.
3. View the marks for the males of class B.
4. What is the average french mark in class C?
5. Display the average mark for each subject, ignoring classes.
6. For class A, produce a table which includes the average, minimum, maximum and standard deviation of the maths marks.
7. Display the lowest, median and highest mark according to gender and subject.
8. How many people are in each class?
9. For class B, produce a table which includes the median, range, sum of the marks and number of distinct marks from the ict marks.
10. Calculate the highest average subject mark for class E, calling it topmark.
11. Without using the **avg** function, calculate the average mark for french in class D. Use the avg function to verify your result.
12. Calculate the range of marks among the males in class D in ict and french.
13. Delete class E's results from the population.
14. It was discovered that Class A's french papers were marked to hard, add 5 marks to each of their scores.
15. Add a new column called average to the table, which contains the average mark for that class and subject.
16. Load the fakedb.q script and create a quotes and trades table.
17. Find the time where the price of each sym, from the trades table, changes by the largest absolute value for the list of syms provided.
18. For an individual sym from the input list, calculate the 10 and 15 trade moving averages. On the table, mark the instances (using boolean) where the values cross.
19. Find the number of times the trade price was below the mid price, above the mid price and equal to the mid price for each sym. Also return the absolute maximum difference between the mid price and the trade price for each sym.

7.5 Hard

1. Load the fakedb.q script and create a trades and quotes table.
2. From the trades table, find the average price of each sym in the five minutes after the hour, each hour, for the entire day. Add to this table the average price of each sym in the last five minutes of each hour.
3. Find a mark to market PNL and a position as well as the volume of all trades before a given time. NB - Assume the trades are from our perspective and use the mid price of the quote at the input time as the market price.
4. Find the price of an index at a particular time. NB - The input should be a dictionary where the keys are the index components and the values are the weights and the return should a number stating the cost of the index.

Chapter 8

Joins

For the following exercises, use the script 'fakedb.q' and the function 'makedb' to create tables.

1. Create a 'quotes' tables (of size 20) and
   (a) Store the first 5 rows as 'q1'
   (b) Store the last 5 rows as 'q2'
   (c) Vertically join 'q1' and 'q2' and store as 'q3'
2. Create a 'quotes' tables (of size 1,000) and
   (a) Calculate 'minbid' (the minimum bid value) and 'maxask' the maximum ask value for each symbol, and store in a separate table
   (b) Left join this table back to the original table
3. Create 'quotes' and 'trades' tables (of size 1,000,000 and 200,000 respectively) and join, to each trade, the prevailing mid price (average of bid and ask) for the relevant time and sym.

Chapter 9

Adverbs

1. If a is a list of lists

```
q)a
103 567
123
459 754 900
```

   (a) Use the count function to determine how many items are in a

   (b) count how many elements are in each element of a

   (c) sum every row of a

   (d) sum the numbers within a

2. Use the line of code:

```
prices:28+.01*floor 100*1000?5.0
```

   to generate a list of trade prices and assume that the order they appear in the list is the order in which they occurred. Find the total price of the largest two consecutive trades.

3. The binomial coefficient $C(n,k)$ represents the number of ways of choosing k elements from a set of n elements. It can be given by the formula

$$C(n,k) = \frac{n!}{k!(n-k)!}.$$

   We can attempt to naively implement this in q by defining:

```
C:{[n;k] prd[1+til n]%prd[1+til k]*prd[1+til (n-k)]}
```

   Using this function find:

   - The value of $C(15,k)$ where k ranges from 2 to 10.
   - The value of $C(n,3)$ where n ranges from 5 to 10.

NB for $n > 20$, this function will give unreliable output as q cannot handle factorials of numbers larger than 20, a more clever implementation is required in this case.

4. Define the following function and variables:

```
q)f:{x+y*8}
q)a:1
q)b:4
q)c:3
q)list1: 1 2 3
q)list2: 10 9 8
q)list3: 3 7 1 5
```

Find which combinations of atoms, lists and adverbs with the function f return the following answers:

(a)

```
35 39 33 37
```

(b)

```
81 82 83
73 74 75
65 66 67
```

(c)

```
81 153 217
```

(d)

```
51 107 115 155
83 139 147 187
35 91  99  139
67 123 131 171
```

(e)

```
129 130 131
```

(f)

```
35 31 57 13
```


Chapter 10

Basic IPC

1. Start a q server on port 1234. Start another q session as a client and define the function f:{x*x}. Execute this function on the q server from the q client with an argument of 7i.
2. Define a function f:{2*x} on the q server. Now from the q client, synchronously run this function on the remote server with an argument of 7i.
3. Repeat 2) but using asynchronous calls.
4. Synchronously, set the value of a variable “h" to 9 on the remote server using both the string form of the ipc call and the parsed function call form.
5. Implement an asynchronous callback in q that would behave in the same way as a synchronous call
6. You wish to send off a batch of asynch messages, but want to be notified that all the messages have been processed by the remote server. How would you do this?
7. You wish to send a number of asynch messages then close down your process. How can you guarantee that all your asynch messages have been sent before you close your process?
8. What is a negative port number used for in q?
9. On the remote server use the .z functions to keep a record of who is logging on from the client servers, and detect when they close their connection.
10. On the remote server, use the .z functions to keep a record of the incoming asynchronous and synchronous calls.
11. How would you serialize a q object to see the byte representation? Give an example.
12. How would you deserialize a message encoded in q ipc format? Give an example.

13. Create an enumerated list.

```
q)u:`c`b`a
q)v:`c`b`a`c`c`b`a`b`a`a`a`c
q)ev:`u$v
```

Retrieve this over ipc - what do you notice?

Chapter 11

Web Access

11.1 Exercises from Notes

1. Start a q server listening on port 9001 and via a web browser complete the following tasks
   (a) Create a table with four fields and 5 entries and name this table 'tab1'.
   (b) Define a function that takes a table as an argument and returns the table in reverse order.
   (c) Define a function that takes a table and an integer as arguments and returns all the records in the table where the index number is less than the integer input.
2. Using the same q server and previously defined table complete the following tasks
   (a) Extract the table as asimple csv file
   (b) Extract the tble as an xml document
   (c) Load the table into Microsoft Excel

11.2 Easy

As the exercise possibilities for the web access chapter is limited, this chapter will also serve as a revision chapter to material covered up to now.

1. (a) Start a q session and type

```
\p 4001
```

   to open the port.

   (b) Create the following table, called 'names', in the q session:

forename	surname	age	gender
jane	brown	32	F
clare	smith	24	F
john	white	61	M
fred	jones	63	M

   (c) View the table in the browser window.

   (d) Using the browser window only:

      i. select all forenames that are female
      ii. add another entry to the table - a 20 year old woman called Anne Smith
      iii. rename Jane Brown as Sarah Brown
      iv. find the average age of all entries in the table
      v. create a function which returns the gender of a given forename from 'names'.

2. Assign the port 6789 to the current q session. In the websbrower:

   (a) execute the 'school.q' script

   (b) view all tables

   (c) select all maths students from class B whose mark is greater than 70

   (d) find the highest mark for each subject for male students

   (e) find the highest mark for each subject for female students

   (f) find the average mark in french and english for female students from class A.

11.3 Moderate

1. Open the script fakedb.q on port 9001 using the command line prompt.
2. View the variables present on a browser window. From this point onwards, only use the browser window.
3. Make a trade and quotes tables of length 1000000 and 200000 respectively.
4. Permanently add a 'diff' column which is the positive difference in the ask and bid prices from the quotes table.
5. How many quotes have a bid-ask spread greater than 0.05 in each symbol?
6. Add an hour onto each of the times.
7. Make a function **f** which will add columns with the average bid price per sym to the table, multiply bidsize by this average bid price calling it result and finally calculate the minimum value of result as the functions output. As well as this, it should let the user know the number of bids for each symbol.
8. Change the port to 9002 using the browser.
9. Data from CSCO is corrupted, the bids and ask are 0.05 higher than should be and the times are half an hour ahead. Correct this data.
10. Remove asks and bids of less than 1000 units.
11. Change the DELL quotes to be AAPL.
12. Create a new keyed table which is keyed by symbol and contains the median ask and bid prices.

Chapter 12

Tables on Disk

For the following exercises, use the script 'fakedb.q' and the function 'makedb' to create tables.

1. Create a directory (e.g. c:/dbex/), with 3 subdirectories 'flat', 'splay' and 'partition'
2. Create quotes and trades tables (of size 10,000 and 1,000 respectively) and
   (a) Store 'quotes' as a flat table, without a name change, inside the 'flat' subdirectory.
   (b) Store 'trades' as a flat table, called 'flattrades', inside the 'flat' subdirectory.
3. Create quotes and trades tables (of size 1,000,000 and 50,000 respectively) and
   (a) Delete any symbol-type columns from 'quotes'. Store as a splayed table (without a name change) in the 'splay' subdirectory.
   (b) Store 'trades' as an enumerated splayed table under the title tradesenum, in the 'splay' subdirectory. Check for a sym file.
   (c) Load up the 'splay' subdirectory in a separate process and select all IBM records from each table.
4. Create quotes and trades tables (of size 1,000,000 and 200,000 respectively) and
   (a) Use .Q.dpft to store 'quotes' to the 'partition' subdirectory, with today's date, and extra partitioning by 'sym'.
   (b) Use .Q.dpft to store 'trades' to the 'partition' subdirectory, with today's date, and extra partitioning by 'sym'.
5. Start a separate process to inspect the partitioned database
6. Start the process listening on port 4321, and inspect the data through a browser (Leave this process running!)

7. Create quotes and trades tables (of size 1,500,000 and 300,000 respectively) and use .Q.hdpf to
   (a) Store both tables to the 'partition' subdirectory
   (b) Use yesterday's date
   (c) Add extra partitioning by 'sym'.
   (d) Inform historical process (on 4321) of the change
8. Check on 4321 that the updates have been loaded

Chapter 13

ETL - Extract, Transform, Load

N.B. The files csvexercise1.csv and csvexercise2.csv are included for the purposes of this exercise.

1. Using a suitable program, view csvexercise1.csv and deduce a suitable description of the table. From this, upload the csvexercise1.csv file appropriately into memory and store as 'quotes'.

2. Upload csvexercise2.csv in the same way, adding column names - time, sym, price, amount and type. Store as 'trades'.

3. In memory, select the AAPL rows and store as a new table 'apples'.

4. Store this table down as a csv file ('apples.csv') in the 'csv' directory.

5. Load 'apples.csv' back into memory, leaving out the 'time' column and storing the 'sym' and 'type' columns as strings.

Chapter 14

Inter Process Communication

These questions relate to kdb+ Inter Process Communication (IPC). IPC is concerned with all aspects of kdb+ processes connecting to each other and communicating. Starting a q process on a port means it can be connected to by other processes. Details on IPC are available here:

http://code.kx.com/wiki/JB:QforMortals2/i_o#Interprocess_Communication
http://code.kx.com/wiki/Startingkdbplus/ipc
http://code.kx.com/wiki/Cookbook/IPCInANutshell
http://code.kx.com/wiki/Reference
http://code.kx.com/wiki/Contrib/UsingDotz
http://code.kx.com/wiki/Reference/ipcprotocol

14.1 Connections

This section is about opening and managing connections. You need to run two q process in the foreground, on different ports e.g.

```
q -p 8000
q -p 8001
```

In these exercises you will be using:

```
hopen, hclose, .z.pw, .z.po, .z.pc, .z.u, .z.a, .z.h, .z.p, -u (command line
    parameter)
```

1. From the process on port 8000, use hopen to open and store in a variable (h) a handle to the process on port 8001. Use any of the forms of hopen.

2. Use the handle to send the string "1+1" to the remote server, and retrieve the result.

3. Use the -u start up flag to restrict the users who can access this process. The users file is a text file containing colon separated usernames and passwords, with each user on a newline. Add two users to the file, joe and bob, with passwords of your choice. Restart the process on port 8001 to pick up the users file.

4. Open a new connection to the remote server, adding the username and password to the connection string. Check the difference between passing a valid and invalid username and password.

5. When a new connection is opened, the username and password checks specified by the -u flag are done first, then .z.pw is invoked, then .z.po (assuming the connection attempt is successful). .z.pw is a dyadic function - the parameters passed to it are the username (as a symbol) and password (as a character array) of the incoming connection. .z.pw returns 1b if the connection is allowed, and 0b if it is not allowed. Modify .z.pw to return true or false depending on whether the incoming username is contained in a list of allowed users. Test it by opening a connection from the client process.

6. .z.po is invoked with the handle of the new connection passed in as a parameter. Create a keyed table called clienthandles which tracks, for each handle, the ip address, username and host of the incoming connection, along with the time the connection was opened. Test it be opening a new connection to the server.

7. When a connection is closed, .z.pc is invoked with the closed handle number. Modify .z.pc to remove the closed handle from the clienthandles table. Test it by using hclose to close the connection from the client side.

14.2 Synchronous Calls

This section is about executing queries and functions in remote processes using synchronous calls. q processes can make synchronous calls using a positive handle number, or asynchronous calls using a negative handle number. Synchronous calls block until an answer is returned, asynchronous calls do not. Both types of call can take multiple forms. Assuming a handle h to a remote server, valid forms are:

```
h"string query"
h(`functionname; param1; param2; ...; param8)
h("functionname"; param1; param2; ... ; param8)
h({lambda}; param1; param2; ... ; param8)
h("{lambda}"; param1; param2; ... ; param8)
```

Every time a q process receives a message, one of the message handlers z.pg or .z.ps are invoked. .z.pg is the synchronous (get) message handler, .z.ps is the asynchronous (set) message handler. Each message handler takes one parameter - the incoming request.

In these exercises in addition to the operations used in the previous exercise you will be using:

```
value, .z.pg, .z.w, -3!
```

1. Define a function

```
add:{x+y}
```


on the server running on port 8001. From the process on port 8000, open a connection and execute "add" in a synchronous call using 3 of the above forms.

2. Define a function

```
multiply:{x*y}
```

on the process on port 8000. From that process, execute multiply on the remote server on port 8001 by sending it in a lambda form with appropriate parameters.

3. On the remote server on port 8001, create a table to store the incoming calls to the process. The table should contain the time of the call, the handle it originated on, the host, ip address and username of the calling process, and the query that has been submitted. The table should also contain an "accepted" boolean column, which will be set to 1b if the process allows the query to be run. Assuming that for this exercise all queries are accepted, modify .z.pg to log the incoming request, evaluate it, and return the result to the client.

4. Modify the synchronous message handler to only accept executions of pre-defined functions. This means that only queries of the form

```
h(`functionname; param1; param2; ...; param8)
```

are allowed and will be executed. All queries should still be logged. If the query is not accepted, the client should be returned an error of

```
'stored procedures only
```

14.3 Asynchronous Calls

This section is concerned with making asynchronous calls. Asynchronous calls are non-blocking and provide greater flexibility than synchronous calls - the server receiving the call is free to execute it when it pleases, rather than being required to return a result to the client as soon as possible. Asynchronous calls can be made to behave like synchronous calls from the client perspective by using deferred synchronous calls. Deferred synchronous calls are when the client sends the request asynchronously, but blocks on the handle to wait for the result. The client process will block until the server returns the result, but the server process has the freedom to execute the client request whenever it likes.

There is another even more flexible method of communication where neither the client nor the server have to block. The crux of it is that the client sends an asynchronous request to the server, which executes it and sends the result back asynchronously to the client, but wraps the result in a function call. The function call should handle the result and undertake any necessary action. This is especially useful in situations where for example the client process has to keep a relatively up-to-date view of some external data set, but cannot rely on the response times of the remote server and/or it doesn't particularly matter how timely the updates are.

1. Open a handle h to the server running on port 8001. Send an asynchronous request of "1+1" using the negative handle. Observe that no result is returned.

2. Create a function called execute which takes a single parameter of the message to execute. The function will execute the message, then return the result back down the handle it originated from (.z.w). The function can be defined either on the server on port 8001, or the client on port 8000 and passed up to the server.

3. From the client, invoke execute on the server using an asynchronous call and any query string / function call you like. There will likely be an error returned to the client.

4. Modify the test to make the asynchronous call, then block on the handle waiting for the result. This is now deferred synchronous.

5. Modify the execute function on the server to queue the requests rather than execute and return them. Use a dictionary of handle!request to store the queue. Create another function, flushqueue, which executes the first item in the queue and returns it to the client. To test, submit a request from the client and block on the handle waiting for the result. Ensure that the request gets queued, the server has full freedom to do whatever it likes, and the client is blocked. Executing flushqueue should return the result to the client, allowing it to proceed.

6. Extend flushqueue to handle errors gracefully. Using the deferred sync mechanism the server cannot return an actual error to the client. Instead, the server has to return some other structure (perhaps an appropriately prefixed error string) which the client has to interpret. Test it by sending a query which will fail.

7. We want to modify the example so the client does not have to block and wait for the result. Create a function on the client side called processresult which for the sake of simplicity will just print out the returned result.

```
processresult:{-1(string .z.z)," the result has been received: ",-3!x}
```

   Both execute and flushqueue need to be extended to allow the client to specify the name of the function that it wants the result called back on. You should add an extra parameter to the execute function. flushqueue will have to wrap the returned value in the supplied function call. Test it by sending in an asynchronous request from the client, but not blocking on the handle .e.g.

```
(neg h)(`execute;`processresult;"1+1")
```

8. Modify .z.pc to remove requests from the request queue if the client closes the connection.

9. Asynchronous calls are not immediately flushed to the socket for sending. Asynchronous calls are queued, and flushed at the next available opportunity. To demonstrate, we can create an example which sends async messages in a loop:


```
do[10;(neg h)"-1 string .z.p"; system"sleep 1"]
```

On the server side, we would expect to see 10 print statements, each approximately 1 second apart. However, what we see is all 10 statements in a row, and they are only fired after the do loop has completed on the client side e.g.

```
2013.12.18D21:42:30.955301000
2013.12.18D21:42:30.955331000
2013.12.18D21:42:30.955343000
2013.12.18D21:42:30.955354000
2013.12.18D21:42:30.955365000
2013.12.18D21:42:30.955376000
2013.12.18D21:42:30.955386000
2013.12.18D21:42:30.955397000
2013.12.18D21:42:30.955407000
2013.12.18D21:42:30.955418000
```

Modify the example code to flush the handle after each send, so each message is executed 1 second apart.

Chapter 15

HDB Queries

These questions should be attempted after completing basic training. To do these exercises you will need to create a historic database. Load fakedb.q and invoke makehdb e.g. makehdb[`:hdb;10;100000;10000]. Then start a new q session and load the HDB e.g. q hdb.

Each question has a difficulty rating. Generally none of the solutions will require a lot of code. A "HARD" question will require thought but not necessarily more code than an "EASY" question.

15.1 Selects

All of these questions are aimed at selecting out data in different ways. The operators and keywords you might use in this section include:

```
select, from, by, where, in, within, =, +, -, >, <, >=, <=, *, %,
$(cast), $(if-else), '(signal), differ, eval, string, enlist
```

1. EASY Write a function called tradeticks which takes 3 parameters of start date, end date and symbols. The function should extract the trade data for the date range and symbol list.

   ```
   tradeticks1:{[startdate; enddate; symbols] ... }
   ```

2. EASY Create a new function, tradeticks2, which has two extra parameters - start time and end time. The function should extract only those trades which fall within the time range i.e. if start time is 12:00 and end time is 14:00, it will only return trades which occur in that window. HINT: you will have to extract the time portion from the timestamp field i.e. time.time or `time$time

3. MED Create a new function, tradeticks3, which is the same as tradeticks1 but uses a start timestamp and end timestamp as the parameters. It should query across the date boundaries. For example, if you run a query from 2012.01.01D09:00 to 2012.01.05D08:00, it should include all the ticks within that period. Make sure

that it uses the date partitions correctly - the date portion should be run first and separately from the timestamp query.

HINT : for questions 4 and 5, try to call functions already written if possible

4. MED Create tradeticks4 which is an optimized version of tradeticks3. If the start timestamp is equal to the first second of the day, and the end timestamp is equal to the last second of the day, then it doesn't execute the where filter on the time column.

5. HARD Create tradeticks5, which is as tradeticks4 except it additionally checks the type of the input. If a date type is passed in for both start date and end date, then it only executes the where on the date column. If the start and end values aren't of the same type (either both timestamps or both dates) then it should throw an error.

6. MED Create a function, quotechanges, which selects from the quote data for a given date list and symbol, only the data where either the bid price changes or the ask price changes.

7. HARD Create a function which allows you to accumulate together bid quantities from the depth table. For example

```
accumdepth:{[d;s;l] ...}
```

will return all ticks for dates d and symbols s. Columns returned will be date, time, symbol, and accumdepth which is the depth sizes accumulated, specified by the parameter l. So if l is 2, accumdepth will contain bidsize1+bidsize2, if it is 3 it will be bidsize1+bidsize2+bidsize3. You will need to use a functional select for this.

15.2 Aggregations

All of these questions are aimed at aggregating data sets, i.e. reducing the raw tick values to aggregative measures. This is generally done with the by clause. The operators and keywords you might use in this section include all those from the previous section but also:

```
update, avg, first, last, max, min, wavg, deltas, xbar, next,
prev, ^, fills, cross, #, lj, til, abs, dev, rank, xrank, ?(find)
count, asc, each, \:(each-left), fby
```

1. EASY The spread for a quote is defined as the difference between the bid and ask price. The ask price should always be higher than the bid price. Write a function which calculates the average spread between bid and ask per date and symbol from the quote table for a given date range and symbol list.

```
avgspread:{[startdate;enddate;symbols] ... }
```


2. EASY A common operation is to calculate High price, Low price, Open price, Close price values for each stock for each day. Create a function, dailystats1, which calculates the HLOC values for a given date range and list of instruments

```
dailystats1:{[startdate;enddate;symbols] ... }
```

3. EASY The VWAP is the Volume Weighted Average Price. kdb+ has a built in function, wavg, for calculating weighted averages. Create dailystats2, which is similar to dailystats1 but also calculates the VWAP value.

4. EASY The date can also be bucketted by time period using the xbar function. Create dailystats3 which is as dailystats2, but includes a bucketting value of type timespan.

```
dailystats3:{[startdate;enddate;symbols;bucket] ... }
```

5. MED Create dailystats4, which is the same as dailystats3 but make sure the OPEN price returned in each bucket is actually the open price. The opening price should always be the same as the close price of the previous bucket for the the same symbol. Is it?

6. HARD Create dailystats5, which is the same as dailystats4 but uses a starttimestamp and endtimestamp instead of startdate and enddate. You need to create a fully contiguous timeseries. If there is no data in a time bucket, then dailystats4 will not return any data. dailystats5 will include all time buckets. This is done by creating a "rack" table which contains all the combinations of symbols and required timebuckets, then filling against it. In this case, the value for each unpopulated HLOC value should be the same as the close price for the previous bucket (as the closing price carries forward).

7. MED The TWAP (time-weighted average price) is similar to the VWAP, but instead of weighting the price by the size, the price is weighted by the length of time it existed as the last price. Create a function, vwapandtwap1 which calculates the VWAP and TWAP per day and per symbol for a supplied list of dates and symbols.

```
twapandvwap1:{[startdate;enddate;symbols] ... }
```

8. HARD Create twapandvwap2, which is as twapandvwap1 but calculates the difference on a percentage basis between the TWAP and VWAP. For each symbol, return the date which shows the biggest differential on an absolute basis. HINT: You could use an fby here

9. MED Create bigdays, which returns across a one month window any days where the volume of shares traded was greater than n stddevs from the average for that month.


```
bigdays:{[mnth;stddevs] ... }
```

10. HARD Create a function, iqr, which takes a list of data and calculates the interquartile range. (Don't have to be too precise about the calculation of the quartiles - an approximate number will do). Use the iqr function to the interquartile range of price for all symbols on any given day.

```
iqrs1:{[startdate;enddate;symbols] ... }
```

11. EASY Create iqrs2, which will calculate for a date range and each symbol, which day had the largest iqr.

12. MED I want to xbar some data in 25 minute buckets. However, my trading day doesn't start until 9am and I want to make sure that 9am -> 09:25am is my first bucket. How can I do that? Create a function to do it. Note that in this function the bucket value should be an int or long rather than a timespan like it was in the previous questions.

```
bucket:{[startdate;enddate;starttime;endtime;bucketsize;symbols] . }
```

13. MED I want to bucket some data in variable width buckets. My bucket start points are 08:00 08:15 08:32 08:50 09:27 12:00. How do I do that? (hint - use bin)

```
variablebucket:{[startdate;enddate;symbols;buckets] ... }
```

14. HARD Create a function to split the trading day into even volume buckets. So for example, if we wanted to split each trading day into 10 even buckets in terms of cumulative quantity traded, whow much volume would be traded in each bucket and what would the start and end times of the bucket be? HINT: look at xrank. Also, only write it to handle 1 symbol at a time.

```
volumebuckets1:{[startdate;enddate;symbol;numbuckets] ... }
```

15. . MED Create volumebuckets2, which calculates the average start time and end time for each volume bucket across a set of dates. This will give the average volume profile. It should invoke volumebuckets1.

```
volumebuckets2:{[startdate;enddate;symbol;numbuckets] ... }
```

15.3 Calculated Data

These questions are aimed at taking raw data sets, and appending new calculated values to it. The operators and keywords you might use in this section include all those from the previous section but also:

```
xprev, sum, sums, bin, flip, &, sublist, exec
```


1. MED Use tradeticks1 to select some data. Create a function to add in a column which computes the return of a price compared with the nth previous price.

```
calcreturns:{[startdate;enddate;symbols;n] ... }
```

2. MED Using the tradeticks1 function, create a function to extract data and add in running vwap and running twap columns. The running VWAP is the (running sum of the price*size)/running sum of size. The running TWAP is the (running sum of the price*active time)/running sum of active time.

```
runningvwapandtwap:{[startdate;enddate;symbols] ... }
```

3. MED Create a function to calculate the running sum of volume traded in each "price group" block. The function should use tradeticks1 to extract data from the trade table. A price group block is defined as a series of trades which are all executed at the same price.

```
pricegroups:{[startdate;enddate;symbols] ... }
```

4. MED Extract a timeseries of data from the depth table (any date, symbol, time range you like). Add a column which displays the VWAP of all the levels on the bid, i.e. the VWAP you would get if you were to clear the depth for each update. Do the same for the ask.

```
sidevwaps:{[startdate;enddate;symbols] ... }
```

5. HARD Extract 30 minutes worth of depth ticks for any date and any symbol. Assuming I wish to sell 1000 shares, calculate for each tick how many levels down the book I will have to go to fill the order.

6. HARD Similarly to above, assume I wish to trade 1000 shares. For each depth tick, calculate the average price I will receive for this trade size.

7. EASY The following function

```
last5:{{-5 sublist x,y}\[();x]}
```

   can be used to return the last 5 values at each point in a list. Try it on a list of integers and observe the result.

   Use the tradeticks1 function defined and add a column containing the last 5 traded prices for each symbol.

8. MED Create a function to create a "rack" of data - a complete timeseries. This should take 4 parameters - a start timestamp, an end timestamp, a list of symbols, and a time bucket (type timespan). Use rack to select a complete bucketed timeseries from the trade table


```
rack:{[starttimestamp;endtimestamp;symbols;bucket] ... }
completeseries:{[starttimestamp;endtimestamp;symbols;bucket] ... }
```

9. HARD Create a function to pivot the following table :

```
t:([] date:2010.01.01 + til 6; sym:`A`B`C`A`B`C;
          price:10 11 15 20 4 11)
```

such that we have a matrix where date is the row identifier and symbol is the column identifier. (HINT : can build this up ourselves, or refer to code.kx.com)

15.4 Joins

This section is about joining data between different tables. The operators and keywords you might use in this section include all those from the previous section but also:

```
?(vector conditional), lj, cor, aj, uj, xcol, `g#(attribute)
```

1. MED Create a function which takes as parameters a list of dates and a list of symbols. The function should extract the trade data and aj on the quote data, then calculate the number of trades which occurred on the bid, those which occurred on the prevailing ask, and those which occurred outside of the bid and the ask for each day.

```
whichside:{[datelist;symbols] ... }
```

2. MED Using the whichside function, create a function which calculates for any set of symbols symbol and over any list of days, which day had the largest disparity between the volume of shares executed on the bid and on the ask.

```
tradedisparity:{[datelist;symbols] ... }
```

3. HARD Create a function, adv, which takes 4 parameters - start date, end date, bucket size as a int (representing minutes) and symbol list. The function should calculate the average daily volume (cumulative) profile for the supplied parameters. Using this function to compare the trading which occurred yesterday with the previous 20 trading days. Calculate, for each time bucket, the percentage of volume that had executed compared with the daily profile.

```
adv:{[stardate;enddate;bucket;symbols] ... }
comparetoadv:{[startdate;enddate;bucket;symbols;comparisondate] . }
```

4. MED Using the adv profile defined above, find the days within any 10 day period where the total volume traded by 12:00 was higher than the ADV for that period.

```
higherthanadv:{[startdate;enddate;symbols;comparisontime] ... }
```


5. MED For one days worth of tick data, calculate the correlation of prices between two stocks which are supplied as a parameter. (this will require lining up the timeseries in some way)

```
pricecorrelation:{[d;symbol1;symbol2] ... }
```

6. HARD For each trade in the trade table on any given date, find the change in price for that symbol in the 5 minutes preceding the trade

```
priceprev5:{[d;symbols] ... }
```

7. HARD For each trade in the trade table on any given date, find the vwap (for the symbol) for the 10 minutes following the trade

```
vwapplus10:{[d;symbols] ... }
```

8. MED We are going to try to create a table containing all the bids / asks at +/- 1 2 5 10 minutes for every trade. Write a function to shift from quote the data at + x minutes. So, for example, if we were to query

```
shiftquote[quotedata;3]
```

   it would return columns symbol, time,bidp3,askp3

   where bidp3 and askp3 are the bid and ask in 3 minutes from the time column.

   and if you were to query

```
shiftquote[quotedata;-3]
```

   it would return columns symbol,time,bidm3,askm3

   where bidm3 and askm3 are the bid and ask from 3 minutes previous.

9. HARD Use aj and / (over) to retrieve, for each trade, all the quotes at +/- 1 2 5 10 minutes. Use the shiftquote function

```
plusminusquote[date;symbols;shifts]
```


Chapter 16

Tickerplant

These questions relate to the setup of a standard kdb+ tick environment. There is documentation available which explains the architecture and set up of a standard kdb+tick environment.

16.1 Tickerplant Setup

In this section we will set up the tickerplant and RDB.

1. Download kdb+ tick. Unzip.

2. Create a schema file containing 3 tables called trade, quote and fxquote. The columns should be:

```
trades:   time,sym,price,size
quotes:   time,sym,bid,ask,bsize,asize
fxquotes: time,sym,bid,ask
```

   Use appropriate types. Use timespan, float and float for times, prices and sizes respectively. The file should be called marketdata.q

3. Start up the tickerplant and RDB in separate process windows. There are examples at the end of tick.q as to how to start each of these. The parameters to the tickerplant should be marketdata (the name of the schema file) and hdb (the directory where the data will be written to at end-of-day). The parameter to the RDB is the tickerplant port. The tickerplant should be started first as the RDB connects to it.

4. Ensure the tickerplant and RDB are running correctly. The tickerplant should have created a log file in the DST directory e.g. ./hdb/marketdataYYYY.MM.DD. The RDB should have connected to the tickerplant, retrieved the schema and replayed the log file. The RDB should have each of the trades, quotes, and fxquotes tables defined.

16.2 Feedhandler

The section contains questions on the feedhandler, which pushes data into the tickerplant. The example feedhandler is based on ssl.q, which is the standard RMDS feedhandler availble on code.kx. Some of the technigues apply to many types of feedhandler or operations in kdb+.

1. Execute createmsglog.q to create a dummy message stream. Change the parameters if you wish, but the defaults should be fine.

2. sslwrapper.q loads ssl1.q, reads in the message log file, and pushes the updates to the tickerplant on a timer. Use sslwrapper.q to start up a feed and push data into the system. It takes 3 parameters - the name of the raw message log file, the host:port of the tickerplant, and the number of updates to push in on each timer loop e.g.

```
q sslwrapper.q rawmsgs :5010 100 -t 1000
```

3. Check the data is being pushed correctly to the tickerplant and on to the RDB. Neither should be throwing any errors. The tables in the RDB should be becoming populated.

4. Restart the RDB and ensure it recovers all the data from the tickerplant log. It should be possible to stop and start the RDB and it will do a full recovery.

5. The src column (the venue which provided the trade or quote) is available on FID 54. We want to add this to the schema. To do that you will need to stop the feed, modify the schema (add the src field as a symbol to each of the trade and quote tables) and modify ssl1.q. You will need to add FID 54 to the list of FIDs for trades and quotes, in the correct position. To pick up the schema change you will need to stop the tickerplant and RDB, remove the tickerplant log, and restart all the processes.

6. The time column is currently being stamped on by the tickerplant. This is because the tickerplant checks the incoming data and if the first column is not of the correct type, it stamps on the current time. Change it so the time column is populated from the feed. We can do this by adding the time FID (18) from the message. However, the time field must be added in the correct place - the schema expects time,sym to be the first two columns in the supplied table.

7. Modify the feedhandler to push the fx data into the correct table - fxquotes. You will need to check for symbols of EURUSD, GBPUSD,AUDUSD. The fxquotes table has columns of time, sym, bid and ask. You can roughly follow the example of how the quote data is currently populated.

8. The quote table has missing values. This is because when the vaule doesn't change, it is not re-broadcast. In the database, we would like to have each row

fully populated, i.e. carry forward the previous value. This can be done at query time, but it is usually better to do it in the feed. Modify the feedhandler so that the quote values are populated correctly, i.e each row contains all values - bid, ask, bid size, ask size. You will need to create a cache function to store and retrieve the previous values.

9. Change the type of the sizes from floats to ints. The bid size is FID 30, ask size is 31 and trade size is 178. Do not remove the tickerplant log file when you restart the processes. The RDB will throw type errors when it tries to replay the log file. Modify the r.q script to catch the error and continue rather than dumping it out to the screen.

16.3 RDB and HDB Rollovers

1. Use the fakedb.q script to write down some historic to the hdb/marketdata directory. Remove the current days data.

2. Create an hdb process which looks at the on-disk data, and run it on port 5012. Check the last date that it has available to it, and the list of tables.

3. The RDB can take one or two command line parameters. The first is the tickerplant port, the second is the HDB port. Restart the RDB so it has the second parameter populated with the HDB port.

4. The RDB will rollover (move all its data from in-memory to on-disk) on a daily basis at midnight local time. The function used to do this is .u.end[currentdate]. The end of day rollover can be simulated by calling .u.end[.z.D] directly. Run this function in the RDB, and check the data is written down to a new date partition. Check that the HDB has been correctly reloaded by the RDB - the last date in the HDB should now be the date that has just been rolled down. Check the list of tables that the HDB now has available in it.

5. Try to run a "select count i by date from table" statement on each of the tables (trades, quotes, fxquotes) in the HDB. Some of them will fail. Why do you think that is? You need to fix the problem - look at the .Q.chk function on code.kx.

6. Reload the HDB (either restart the process or run

```
\l .
```

to reload). Check the list of tables available.

7. We can't fix up the database everyday as it is too time consuming. To make sure we don't have to do all this again, we need to add depth to the schema - so on future roll overs, the depth table will be populated with blank data. Restart the tickerplant, RDB and feedhandler to ensure the schema changes are picked up. Test the change by rolling over the RDB to a dummy date e.g. .u.end[2100.01.01]

and ensuring the HDB can access all the tables. You can remove the 2100.01.01 date when done and reload the HDB.

8. Run a query of

```
select by date from trades
```

What do you notice? You should see that there is a difference in the time column. The data which was created by fakedb.q has time stored as a timestamp, whereas the data which came from the feed has a timespan type. We need to make the types the same. One place we can do this is in the end of day function, .u.end. Modify it in r.q to make the time column of each table a timestamp rather than timespan. When done, re-run .u.end and check the data has been correctly updated.

9. It may be the case that you need to modify the more of the historic data if there has been multiple dates written down with the incorrect type. If you need to do this, look at the dbmaint.q script available on code.kx. This script allows adjustments to historic data to be made.

10. A usual operation to execute at the end-of-day roll down is to save some additional data which is commonly required. An example is an end-of-day high, low, open and close price for each instrument. This is usually one row per instrument and date, saved as a splayed table in the top level HDB directory. Examples of this are available in daily.q on code.kx. Modify .u.end to additionally store down the daily table. Remember to set the partition flag on the date column.

16.4 Real Time Analytics

This section is about creating custom scripts to connect to the tickerplant and subscribe to data, then doing calculations on that data.

Any process can subscribe to the tickerplant. Subscriptions are done on a table and symbol basis, i.e. subscribe to a list of tables and a list of symbols. The null symbol is used to subscribe to all symbols or all tables. A standard subscription request to the tickerplant would be

```
.u.sub[list of tables; list of symbols]
```

e.g.

```
.u.sub[`trades`quotes;`IBM`NOK`AAPL]
.u.sub[`;`]
```

When the process has subscribed, the tickerplant will send messages to it of the form

```
(`upd;`nameoftable;tabledata)
```


To handle the message, the subscriber must have a function called upd defined which takes two parameters - the name of the table as a symbol and the table data. The subscriber can do anything it likes with the data in the upd function.

1. Create a process which connects to the tickerplant, subscribes to all trade data, and maintains the total count of each instrument received. The process should take one parameter of the host:port of the tickeplant to connect to. Errors should be handled appropriately, e.g. if no connection can be made to the tickerplant the process should exit.

2. On the tickerplant process, inspect the .u.w variable. This holds the subscription lists.

3. The running vwap can be calculated by tracking the total traded size and the sum of the price*size for each instrument. The VWAP at any point in time is then (sum of price*size)/(sum of size). Modify the subscriber to track the VWAP. You will need to create an appropriate table to store the necessary information, keep the table up to date using the upd function, and have a calcvwap[list of instruments] function which can be invoked to calcuate the VWAP.

4. If the RDB fails it recovers by replaying the tickerplant log file. The subscriber could also recover by replaying the tickerplant log file. However, a faster but approximate method for recovery is to query the RDB for the relevant data (it is an approximate method as it is possible to miss data or duplicate data, depending on the sequencing). Add an extra command line parameter to the subscriber which is the host:port of the RDB. On start up, the subscriber should query the RDB to get the current trade counts and VWAP data.

5. Every process can also be a publisher of data. The u.q script can be used for publishing. u.q should be loaded, then

```
.u.init[]
```

   invoked after all the tables to be published have been declared in the top level namespace. The function

```
.u.pub[nameoftable; tabledata]
```

   does the publishing. Modify the subscriber process so that on each update it publishes out the updated VWAPs in an appropriately named and timestamped table Any table to be published requires a column called sym.

6. You need to test that the publication is working. Create a test process (no need for error traps etc.) to connect to your subscriber and receive the VWAP data. It is sufficient just to print the updates to the console.

16.5 Daemonizing

This section is about creating scripts to start and stop the system, and to get processes to run in the background (in the case of unix type systems).

1. Write a script to start all the processes. If the operating system is windows, this will be a batch file (.bat) otherwise it will be a shell script (.sh).

   If windows, the start lines will look like

   ```
   start "NAME OF PROCESS" startcommand
   ```

   e.g.

   ```
   start "Tickerplant" q tick.q marketdata hdb -p 5010
   ```

   An example linux start command is given at the top of tick.q i.e.

   ```
   q tick.q marketdata hdb -p 5010 </dev/null >tickerplant.log 2>&1 &
   ```

   Execute the script and make sure all the processes have started correctly (using either task manager or a ps command). Check the log files for errors.

2. Write a kill script to remotely kill a process. The kill script will take a single parameter, which is the port of the process to be killed. It will create a connection to that process and send an exit command before killing itself. Remember to use error traps and logging appropriately. For example, if the connection can't be opened, the error should be caught and logged. If the exit command is sent asynchronously, the handle should be flushed before the kill process kills itself.

3. Write a batch script or shell script to kill each of the tick environment processes.

4. A better solution might be to modify the kill script to accept a list of host:ports to kill. Can you do that?

Chapter 17

File Loading

These question relate to loading data from flat files. All the questions centre around loading data from csv (comma separated value) files. They will cover topics including dropping columns, manipulating columns, renaming columns, sorting data in memory and on disk, loading files in chunks to preserve memory, saving tables in different formats (partitioned, splayed, flat). References can be found on `code.kx.com`, specifically

`http://code.kx.com/wiki/Cookbook/UsingKdb#How_do_I_import_a_CSV_file_into_a_table`
`http://code.kx.com/wiki/Cookbook/LoadingFromLargeFiles`
`http://code.kx.com/wiki/Cookbook/LoadingFromLargeFilesAndSplaying`
`http://code.kx.com/wiki/Cookbook/SplayedTables`
`http://code.kx.com/wiki/JB:KdbplusForMortals/partitioned_tables`
`http://code.kx.com/wiki/Reference/set`

17.1 Basic Loading

1. Use the fakedb.q script to create some csv files e.g.

```
makecsv[`:csv;5;1000000;100000]
```

2. Create a function to load one of the csv files into a table in-memory. Use the 0: operator and use appropriate types. Load all the columns. The function should take a single parameter - the full path to the csv file to load.

```
load1file:{[csvfile] ... }
```

3. Modify the function to ignore the trade flag column.

4. Modify the function to rename the columns to appropriate names. kdb+ tables should not contain column names with spaces, and the column naming should be consistent with other data sets (e.g. INSTRUMENT should be sym). HINT: use xcol.

5. The EXCH column currently has some superfluous extra information in it. The "EXCH=" string is not required on each value. Modify the loading function to drop this extra data.

6. We want to store the time column as type timestamp. However, there isn't any date information in the data - only in the file name. Modify the function to extract the date from the filename and convert the time column to timestamp.

7. We want to store the trades and quotes into separate tables, in a date partitioned HDB. Modify the function to split the input table into two tables, and save to the correct date partition. Use .Q.dpft, which will require the tables to be set as global variables. For trades, store time, sym, price and size, and for quotes store time, sym, bid, ask, bsize and asize. A trade is indicated by the price field being non-null. The function should now take two parameters - the hdb directory and the csvfile e.g.

```
load1file:{[hdbdir;csvfile] ... }
```

8. Tables can be stored as partitioned (each partition directory contains a different set of data for each table), top-level splayed (the table is splayed but only exists in a single directory at the top level of the database) or binary (a kdb+ binary object which exists in the top level HDB directory and is loaded into memory when the HDB starts up). We would like to store two additional tables:

   - A table called static, which is keyed by sym and stores the static data for each instrument. In our case, this is just the currency value. This should be stored as a binary table.
   - A table called daily, which has one row per date and sym, and stores some daily aggregative values for each instrument. We want to store the open price, close price, high price and low price for each instrument.

   Modify the function to store down these two tables.

9. Execute the loading function with \ts and observe the time and memory usage.

17.2 Loading Big Data Sets

This section is concerned with loading datasets without having to read the whole dataset into memory. It will centre around the .Q.fs and .Q.fsn functions. .Q.fs and .Q.fsn allow a file to be read in chunks, rather than reading the whole file into memory at once.

1. Remove the header line (column names) from each of the csv files created in the previous exercise.

2. Create a function to read in the file as before, and write out the trades and quotes tables to a partitioned database. The function should be similar to the solutions for the previous exercise set, but with several differences:

- The function should take 3 parameters - the hdb directory, the name of the csv file and the csv data.
- When the data is read in, the column names will have to be manually added to the data.
- Do not use .Q.dpft to write the data - instead use a method similar to that used to write out the daily table in the previous question set.
- Ensure the code prints debug information which allows you to monitor the progress of the operation.
- Do not write out the static or daily tables.

The function should have a signature of:

```
load1filechunk:{[hdbdir;csvfile;csvdata] ... }
```

3. Measure the time and memory usage for the operation.
4. Use .Q.fs to load the file in chunks.
5. Measure the time and memory usage for the operation.
6. You should see that when running with .Q.fs the memory usage is smaller as small chunks of data are read in, but the total time is increased due to more individual operations rather than one big operation. Experiment with .Q.fsn to find the optimal size of data to read in at once - so the memory usage is lower, but the total time isn't significantly different to reading the file in one big batch.
7. Create a wrapper function to invoke the chunk loader within .Q.fsn for a given file e.g.

```
load1bigfile:{[hdbdir;csvfile] ... }
```

8. The data saved down by load1bigfile is not the same as that saved by load1file. The data saved by load1bigfile is parted by date within each date partition. Modify load1bigfile to sort the data on disk, then set the attribute after all the data has been written to disk.
9. Create a function to load all the csv files in a specified directory. The function should use load1bigfile.

```
loadcsvs:{[hdbdir;csvdir] ... }
```

10. Can you see any potential problems with the above scripts w.r.t. duplicate data?

Chapter 18

Creating a Trade Report

This training exercise will have a number of parts, with the end goal of creating a script that will query a database and generate a report as a csv file. As a prerequisite you will need to start a q process, create an hbd using the fakedb.q script, and set the process listening on a port (we will use the port 2222). You should also download tradereport.zip and extract it into your working directory. The tradereport folder contains static data csv files that the script will need to load.

Question
Create a q script that will generate a report, according to the specification below, showing the number of trades, volume traded, and turnover in USD for each sym and each date within a specified date range.

18.1 Report specification

- script must accept start and end dates as command line parameters. Execution example: `q tradereport.q -dates 2014.04.21 2014.05.01`
- script must exit with an appropriate error message if it fails to connect to the HDB.
- table columns required: date, sym, description, curr, ntrades, size, turnover (sum of price x size)
- table should be saved as a csv file 'tradereport.csv'.

18.2 Tasks

1. Begin the script by defining the variable `dates` as a vector of the start and end dates from the command line parameters, ensuring they are of the correct type. Save your script and run it, passing in some dates as command line parameters and check that `dates` is correctly defined.

2. Add a protected evaluation of hopen to open a connection to the HDB process and assign it to the variable `h`. If the hopen fails, the script should return an error message "Failed to connect to HDB:" along with a string of the error returned. Save your script and test your code with both the correct and an incorrect port.

3. Add a query of the HDB via IPC, returning ntrades (number of trades), total size traded, and turnover by date and sym from the trades table. Don't worry about currency at this point. Pass in the `dates` variable as the range of dates to query. Assign the returned table to a variable `trades`. The table should look like the following:

```
q)trades
date       sym  ntrades size   turnover
-------------------------------------------
2014.04.21 AAPL 98      227975 5810067
2014.04.21 CSCO 92      247516 8759907
2014.04.21 DELL 91      251657 7276283
2014.04.21 GOOG 129     330564 1.364547e+007
2014.04.21 IBM  117     284233 1.248481e+007
..
```

4. Load the symstatic.csv (path `` `:tradereport/symstatic.csv ``) into a table `static`. Column types for sym, description and curr should be symbol, string and symbol respectively.

```
q)static
sym  description          curr
-----------------------------
YHOO "Yahoo! Inc."        EUR
GOOG "Alphabet Inc."      EUR
NOK  "Nokia Corp (ADR)"   USD
ORCL "Oracle Corporation" GBP
AAPL "Apple Inc."         USD
..
```

5. Join `static` onto `trades` (key/unkey the tables as necessary) and save the results to `trades`.

6. Load the fxrates.csv into a table `fxrates`. Column types should be date, symbol and float respectively.

```
q)fxrates
date       curr rateUSD
----------------------
2014.04.20 EUR  1.1343
2014.04.27 EUR  1.1345
2014.05.04 EUR  1.1339
2014.05.11 EUR  1.1334
2014.05.18 EUR  1.1338
..
```

7. Join the prevailing fxrate by date and curr from `fxrates` onto `trades`, again assigning the result to `trades`. Note that `fxrates` only contains fx rates for

EUR and GBP. Set `rateUSD` to be 1 where the sym's currency is already in USD.

8. Create a `tradereport` table by selecting the required columns from trades and converting the turnover to USD.

```
q)tradereport
date       sym  description             curr ntrades size   turnoverUSD
-------------------------------------------------------------------------
2014.04.21 AAPL "Apple Inc."            USD  98      227975 5810067
2014.04.21 CSCO "Cisco Systems, Inc."   GBP  92      247516 12331758
2014.04.21 DELL "Dell Inc."             GBP  91      251657 10243187
2014.04.21 GOOG "Alphabet Inc."         EUR  129     330564 15478059
2014.04.21 MSFT "Microsoft Corporation" USD  120     286254 10334677
..
```

9. Finally, include a line in the script to write the `tradereport` table to a csv file in the tradereport directory, and end the script with `exit 0`.

18.3 Optional Extras

1. Add logging messages consisting of the current timestamp and a brief descriptive message directed to STDOUT for steps 2-9 above, so that when executed it outputs something like the following:

```
q)2016.04.11D13:28:25.155607000: Connecting to HDB...
2016.04.11D13:28:27.115719000: Querying HDB...
2016.04.11D13:28:38.579375000: Loading symstatic.csv...
2016.04.11D13:28:38.579375000: Joining static data...
2016.04.11D13:28:38.580375000: Loading fxrates.csv...
2016.04.11D13:28:38.580375000: Joining fxrates...
2016.04.11D13:28:38.580375000: Writing to csv...
```

2. Modify the script to save the csv file as `tradereport_YYYYMMDD_YYYYMMDD.csv` where the dates appended are the start and end dates provided.

