Price Impact Models and Applications

Introduction to Algorithmic Trading

Kevin Webster

Spring 2023

Columbia University

Plan

Last Week
Preview of the two modules, trading data.

For this Week
An introduction to kdb+ and q:

(a) Setting up q
(b) Hello world!
(c) Loading a database
(d) Data grammar and basic data manipulation
(e) Advantages and pitfalls of q

Next Week
Mathematical foundation of price impact models.

Kdb+ Documentation (Recommended)

Kx website

(a) references: `code.kx.com/q/ref`

(b) course: `kx.com/academy`

(c) guide for quants: `code.kx.com/q/learn/brief-introduction`

Introduction

Why Kdb+ and Q?

Novotny et al. (Nomura, 2019)

"The kdb+ database and its underlying programming language, q, are the standard tools that financial institutions use for handling high-frequency data."

Efinancialcareers.com (2019)

"if you want to be assured of a job in finance, it will benefit you to learn the coding languages K and Q. K and Q underpin the Kdb+ database system which is used increasingly by banks, hedge funds and high frequency trading houses"

Efinancialcareers.com (2021)

"Developers proficient in both Q and kdb+, the database system that goes with it, tend to be both hard to find and in constant demand globally."

What are Kdb+ and Q?*

Kx systems develops Kdb+
Kx systems describe kdb+ on their web page "Developing with kdb+ and the q language":

(1) "a high-performance cross-platform historical-time series columnar database
(2) an in-memory compute engine
(3) a real-time streaming processor
(4) an expressive query and programming language called q"

An example

```
select avg slippage by date from orderTable
                            where date > 2020.12.31
```

What do Firms Use Kdb+ for?

Real time trading (3)
Real-time analytics, trading data, algorithms, and signal generation.

Historical replay (1)
Historical monitoring, performance analysis (TCA), and signal research and backtests.

Never underestimate a real-time Graphical User Interface (GUI) for stakeholders:

> *"Though I did ultimately improve the model, the traders benefited most from the friendly user interface I programmed into it. This simple ergonomic change had a far greater impact on their business." (Derman, Goldman 2004)*

Case-study: Almgren's Quantitative Brokers

Real Time Trading Signals, Almgren (2018)
Almgren outlines Quantitative Brokers' trading technology:
`www.youtube.com/watch?v=s4IdoWUhRDA`.

> *"Signal generator (Kdb+)*
> *The signal generator receives market data, performs computations to predict prices, and feeds the results to the algorithmic engine to improve trade execution."*

Essential argument (1)+(3)
Quantitative Brokers uses the same code to generate signals in research and production: signals' reproducibility and accuracy are crucial.

Competitive Edge of Kdb+: Core Performance (1/2)

Native vectorization (2)
Q *natively* optimizes data structures and functions for vector and time series manipulation.

Automatic parallelization (1)+(2)
Q *automatically* parallelizes operations. For example,

```
select avg slippage by date from tbl
```

automatically parallelizes by date.

Competitive Edge of Kdb+: Core Performance (2/2)*

On disk operations (2)
Q can do (some) computations without loading the data in memory. For example,

```
smallTbl lj bigTbl
```

joins bigTbl onto smallTbl even if bigTbl does not fit in memory.

Competitive Edge of Kdb+: Data Locality (1/2)

Data locality is an essential principle for dealing with large datasets.

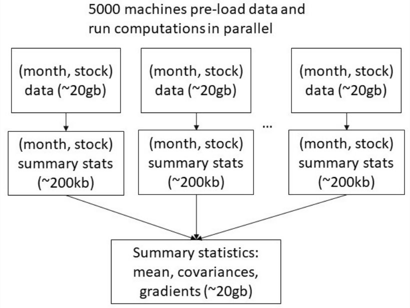
Models should run computations where one stores the data.

Why data locality matters (1)+(2)
Massive data incurs an upfront transfer cost before any computation can start. This data marshaling cost is prohibitive for the iterative coding style of researchers and traders.

Example in Python: `www.dask.org`
The Python package Dask implements Pandas grammar with data locality as a core principle.

Competitive Edge of Kdb+: Data Locality (2/2)

Distribute to local data (1)+(2).
Loading data in and out of memory can be expensive. Data locality pre-loads specific data sections on given servers. The central controller distributes computations to servers where the data is local.



When to Use Kdb+ and When to Use Python?

A prerequisite
Both kdb+ and python must be properly set up to allow for seamless integration with high frequency data.

(a) For kdb+, this involves a distributed database.

(b) For Python, this involves a distribution package such as Dask.

Use kdb+ (or C++)
when you want to be close to production: e.g., GUIs, low-latency signals, and live testing require a close connection to production data to remain accurate.

Use python (or R)
when you want to leverage off-the-shelf machine learning packages: e.g., do you want to code neural networks from scratch?

Most quantitative trading teams mix both low and high-level languages.

First Steps in Kdb+

Download and Install the Personal License

Kx's general introduction
code.kx.com/q/learn/startingkdb

Download the 32-Bit Personal Edition of kdb+
kx.com/developers/download-licenses

By default, q installs in folders:

```
// unix
~/q/l32

// mac
~/q/m32

// windows
c:\q\w32
```

Run Hello World!

Run the executable in a shell.

```
c:\q\w32>q.exe
KDB+ 3.6 2019.04.02 Copyright (C) 1993-2019 Kx Systems
w32/ 16()core 4095MB webst desktop-jfb94cd 10.0.0.213 NONEXPIRE

Welcome to kdb+ 32bit edition
For support please see http://groups.google.com/d/forum/personal-kdbplus
Tutorials can be found at http://code.kx.com
To exit, type \\
To remove this startup msg, edit q.q
q)
```

Figure 1: Q's command line.

"Hello World!" tests the installation.

```
show "Hello World!"
```

Save and Run a Script

For sizable scripts, running from terminal is cumbersome.
Two solutions:

(a) Use an IDE: e.g., Qpad, JupyterQ, or Visual Studio with a q plugin.

(b) Write your script in a .q file and load it.

Example for (b)

```
show "Hello World!"
```

Listing 1: test.q

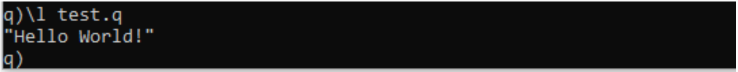


Figure 2: Loading a q-script.

Debug Mode and Exit

Q automatically enters debug mode upon errors.

```
\terminal when not in debug mode
q)
\terminal when in debug mode
q))
```

The backslash command exits debug mode, and double backslash exits q altogether.

Figure 3: Example q error.

Loading Data

Same command as loading a script
will "load" a database folder into your Q session. In practice

(a) An object appears for each table in the database.

(b) You can interact with the object as if it were a table.

(c) The table is not *actually* loaded in memory: it is only "mapped" for fast reading (2).

```
q)\l C:\lobster\hdb
q)tbl: select from daily
q)tbl
date       stock open    close   dailyTrdLiq  eodImb        eodOfi          ..
----------------------------------------------------------------------------..
2019.01.02 A     66.215  65.71   1.88388e+007 -221602       1761854         ..
2019.01.02 AAL   31.525  32.495  3.491806e+007 -394635.8    1.709619e+007   ..
2019.01.02 AAP   156.545 157.965 2.873893e+007 -4987624     2836347         ..
2019.01.02 AAPL  154.915 157.925 8.596159e+008 9295054      1.153468e+008   ..
```

Figure 4: Mapping a database.

Read a File

Reading csvs
The function 0: reads a csv file. See
code.kx.com/q/ref/file-text/#load-csv.

```
\msgFile is the file capturing all the messages on the
    public tape
  msgFile: `$ (string stockName), "_2012-06-21
    _34200000_57600000_message_1.csv";

\The read function requires column types (f is float, j is
    integer).
  msg:  0:[("fjjjjj"; ","); msgFile];

\we manually add column names
  msg: (`time`eventType`orderId`size`price`direction)!msg;
```

Database Properties (1/2)

Databases are faster than csvs.

Databases are larger than what fits in memory
You can think of database object as a pointer with (most) table methods (1)+(2).

(a) Tables only load in memory after an explicit query. Therefore, building concise queries matters (e.g., only load columns you need).

(b) If aggregating data down (e.g. by stock, date), do this *before* loading the data in memory: kdb+ implements common operations (e.g. averages) on disk before returning the result in memory.

(c) Data engineers set attributes onto specific table columns to speed up operations. For example, most tables are *partitioned* by date, *sorted* by (stock, time). They may also guarantee *uniqueness* of certain id columns.

Database Properties (2/2)

```
q)select avg mid by stock from bin10 where date = 2019.02.01
stock| mid
-----| --------
A    | 76.10731
AAL  | 36.28517
AAP  | 159.3925
AAPL | 167.2042
```

Figure 5: Sample database query.

Best practices

(a) Check the table meta information first.

(b) Always start with a date where clause, e.g., date = x or date within (x y).

(c) Only load columns you need.

(d) Consider on disk aggregations.

(e) Consider on disk joins.

```
q)meta bin10
c         | t f a
----------| -----
date      | d
time      | v
stock     | s   g
trade     | j
orderFlow | j
hidden    | j
auction   | j
mid       | f
midEnd    | f
spread    | f
effSpread | f
lobImb    | f
effLobImb | f
trdLiq    | j
ofLiq     | j
depth     | f
nbEvents  | j
nbHidden  | i
nbTrades  | i
```

Figure 6: Table meta information.

Data Grammar in Q

What is a Data Grammar?

Quants manipulate sizable data daily
Modern languages break down data manipulation in small actions (verbs) applied to tables (nouns). The precise syntax defines a language's or package's data grammar and aids readability, flexibility, and re-use of code.

Examples in other languages

(a) An old and well-established data grammar are SQL queries.
(b) The first package to formalize the idea of data grammar is dplyr from R by Hadley Wickham.
(c) The most used grammar in data science is pandas.

Introducing Qsql queries

Documentation
`code.kx.com/q/basics/qsql`

Queries transform tables using two clauses and an action

(a) Where clauses filter data.

(b) By clauses group data.

(c) Actions apply a function to the data.

For instance, the following computes each stock's cumulative sum of trades above 10^4 shares.

```
select sums trade by stock from tbl where abs[trade] > 1e4
```

Joining tables
Two tables merge, typically based common columns called *keys*.

Comparison to Sql queries (4)

Sql is still the standard
The pandas documentation refers to it as a starting point:

> *"Since many potential pandas users have some familiarity with SQL, this page is meant to provide some examples of how various SQL operations would be performed using pandas."*

Qsql mimics sql grammar
The syntax is the same. Q also has a more general but less user-friendly grammar, called functional select. See `code.kx.com/q/basics/funsql`.

Comparison to Python's Pandas

Where clauses are like filters

```
select from tbl where stock = `AAPL

tbl[tbl["stock"] = "AAPL"]
```

By clauses are like groupby

```
select avg return by stock from tbl

tbl.groupby("stock").agg({"return": np.mean})
```

Joins are like merges

```
tbl1 lj `stock xkey tbl2

pd.merge(tbl1, tbl2, on="stock", how = "left")
```

Comparison to R's Dplyr

Where clauses are like filters

```
select from tbl where stock = `AAPL

tbl %>% filter(stock == "AAPL")
```

By clauses are like group_by

```
select avg return by stock from tbl

tbl %>% group_by(stock) %>% summarize(mean(return))
```

Joins are like left_join

```
tbl1 lj `stock xkey tbl2

tbl1 %>% left_join(tbl2, by = "stock")
```

Multiple Where Clauses

Q interprets where clause chains as "and" statements.
Where clauses are (nearly the only thing in q) evaluated left to right: start with the date clause!

```
\good practice
select from tbl where date within (2019.01.01 2019.01.31),
    stock = `AAPL

\bad practice
select from tbl where stock = `AAPL, date within (2019.01.01
     2019.01.31)
```

By Clause Example

Example: binning fill data into 10 second intervals.

Time	Stock	Trade	Midprice	Spread
9:30:01	AAPL	-70	154.77	0.18
9:30:03	AAPL	100	154.74	0.16
9:30:07	AAPL	-100	154.705	0.10
9:30:12	AAPL	234	154.705	0.025
9:30:13	AAPL	-10	154.78	0.02

TimeBin	Stock	Trade	MidStart	Spread
9:30:00	AAPL	-70	154.77	0.18
9:30:10	AAPL	224	154.705	0.10

```
\the pattern (x xbar y) bins the vector y every x units. See
     code.kx.com/q/ref/xbar
select sum trade,
       midStart: first midPrice,
       first stock, first spread
       by timeBin: 10 xbar time from tbl
```

Update vs Select for By Clauses.

Select statements return one row per group

```
q)select avg size by stock from tbl where time < 10:00:00
stock| size
-----| --------
AAPL | 87.62302
AMZN | 86.2926
GOOG | 97.90174
INTC | 372.7082
MSFT | 408.5571
```

Update statements return one vector per group*

```
q)update avg price, sums size by stock from tbl
time         stock size price
--------------------------------
09:30:00.004 AAPL  18   5831.436
09:30:00.026 AAPL  36   5831.436
09:30:00.202 AAPL  54   5831.436
09:30:00.202 AAPL  72   5831.436
09:30:00.206 AAPL  90   5831.436
09:30:00.272 AAPL  110  5831.436
09:30:00.272 AAPL  150  5831.436
09:30:00.275 AAPL  190  5831.436
09:30:00.275 AAPL  215  5831.436
09:30:00.275 AAPL  216  5831.436
09:30:00.275 AAPL  226  5831.436
09:30:00.275 AAPL  251  5831.436
09:30:00.275 AAPL  256  5831.436
09:30:00.275 AAPL  263  5831.436
09:30:00.275 AAPL  283  5831.436
09:30:00.275 AAPL  308  5831.436
09:30:00.275 AAPL  328  5831.436
09:30:00.275 AAPL  428  5831.436
09:30:00.275 AAPL  432  5831.436
09:30:00.275 AAPL  437  5831.436
..
```

Join Operators

Three frequent joins

(a) Comma join joins data by rows or columns.
(b) Left join joins data by keys.
(c) As-of join joins data by the previous timestamp.

Comma Join*

Joining rows
The comma operator appends two conforming data structures. For tables, if tbl1 and tbl2 share columns and data types,

```
tbl1, tbl2
```

appends tbl2 rows after tbl1 rows.

Joining columns
The following pattern joins two tables' columns. Tbl1 and tbl2 must have the same number of rows.

```
flip (flip tbl1), (flip tbl2)
```

Left Join (1/2)

Documentation
code.kx.com/q/ref/lj

Example
Left join merges two tables based on shared columns, called keys. Rows from the two tables with matching keys merge into one row. For instance

```
trades lj `stock`date xkey volatility
```

adds a volatility table to the trades table, matching on stock and date.

Left Join (2/2)

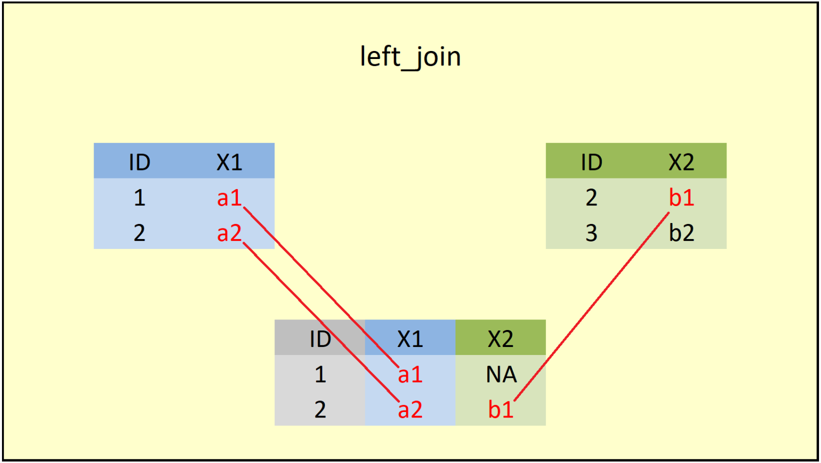


Figure 7: Source: statisticsglobe.com

As-of Join (1/2)

Documentation
`code.kx.com/q/ref/aj`

Asynchronous timeseries are common in trading
For example, let tbl1 represent trades on AAPL and tbl2 trades on GOOG. Trades on AAPL and GOOG are unlikely to happen exactly simultaneously. Therefore, using time as a key variable leads to few matches when using precise timestamps.

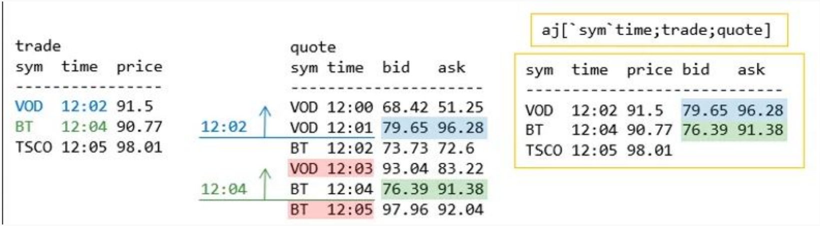
The last trade on AAPL as of the trade time on GOOG

```
aj[`time; tbl1; tbl2]
```

For instance, this may produce

```
timeGOOG        |    midGOOG     timeAAPL     midAAPL
----------------|--------------------------------------
10:01:09.001    |   94.37       10:01:08.967    142.68
10:01:15.127    |   94.36       10:01:08.967    142.68
10:02:05.900    |   94.39       10:02:02.186    141.65
```

As-of Join (2/2)



trade

sym	time	price
VOD	12:02	91.5
BT	12:04	90.77
TSCO	12:05	98.01

12:02
12:04

quote

sym	time	bid	ask
VOD	12:00	68.42	51.25
VOD	12:01	79.65	96.28
BT	12:02	73.73	72.6
VOD	12:03	93.04	83.22
BT	12:04	76.39	91.38
BT	12:05	97.96	92.04

```
aj[`sym`time;trade;quote]
```

sym	time	price	bid	ask
VOD	12:02	91.5	79.65	96.28
BT	12:04	90.77	76.39	91.38
TSCO	12:05	98.01		

Figure 8: Additional example. Source: version1.com

A helpful practice for live trading*
Always add a column that gives the as-of time from the other table: this indicates how "stale" the as-of information is. It is particularly useful when considering algorithms' latency requirements.

Advantages and Pitfalls of Q

Basic Operations (1/4)

A common source of bugs
For performance reasons, q evaluates statements *from right to left, regardless of the standard order of operations*. This property leads to counter-intuitive results.

```
q)1+5*1
6
q)5*1+1
10
```

Figure 9: Counter-intuitive q example.

Basic Operations (2/4)

List of q operators
`code.kx.com/q/ref/`

abs	cor	ej	gtime	like	mins	prev	scov	system	wavg
acos	cos	ema	hclose	lj ljf	mmax	prior	sdev	tables	where
aj aj0	count	enlist	hcount	load	mmin	rand	select	tan	while
ajf ajf0	cov	eval	hdel	log	mmu	rank	set	til	within
all	cross	except	hopen	lower	mod	ratios	setenv	trim	wj wj1
and	csv	exec	hsym	lsq	msum	raze	show	type	wsum
any	cut	exit	iasc	ltime	neg	read0	signum	uj ujf	xasc
asc	delete	exp	idesc	ltrim	next	read1	sin	ungroup	xbar
asin	deltas	fby	if	mavg	not	reciprocal	sqrt	union	xcol
asof	desc	fills	ij ijf	max	null	reval	ss	update	xcols
atan	dev	first	in	maxs	or	reverse	ssr	upper	xdesc
attr	differ	fkeys	insert	mcount	over	rload	string	upsert	xexp
avg	distinct	flip	inter	md5	parse	rotate	sublist	value	xgroup
avgs	div	floor	inv	mdev	peach	rsave	sum	var	xkey
bin binr	do	get	key	med	pj	rtrim	sums	view	xlog
ceiling	dsave	getenv	keys	meta	prd	save	sv	views	xprev
cols	each	group	last	min	prds	scan	svar	vs	xrank

```
q)a:1
q)a=1
1b
```

Figure 10: Variable assignement vs equality.

Basic Operations (3/4)

Terseness
Q scripts are straightforward to *write*.

(a) many high-performance, inbuilt base operators
(b) expressive data grammar, ideal for scripting
(c) overloaded functions mean that related operations “feel” similar
(d) most tasks can be written in a few lines of q

Excessive terseness
It is hard to read q code. Even the original coder will forget what the cryptic one-liner does two weeks from now!

Basic Operations (4/4)

Best practices

(a) when possible, write qsql

(b) when not possible, still write qsql (in a comment)

(c) wrap unreadable q snippets in functions. Document functions

(d) use dictionaries as function arguments when you need extra flexibility (advanced)

Data Types: Atoms

What is an atom?
Atoms represent an indivisible data unit of a single type.

code.kx.com/q/basics/datatypes/.

Casting types in q
One changes an object's type using the $ operator and checks an object's type using the function "type", which returns the type's integer representation.

Figure 11: Type example.

Data Types: Lists

A list is a collection of objects
Vectors are lists of atoms with a single type. Q vectorizes essential functions and operations: when performed on vectors, these functions significantly outperform traditional loops.

```
q)1 2 3 4
1 2 3 4
q)1 + til 4
1 2 3 4
```

Figure 12: Vector example.

Mixed lists are lists of varying types.
One uses mixed lists as function arguments and avoids their use for massive datasets.

Data Types: Dictionaries

Dictionaries take in two lists: keys and values.
Q assumes keys are unique, and the dictionary defines a map from keys to values.

```
q)(`a`b`c)!(1 1 3)
a| 1
b| 1
c| 3
q)dic: (`a`b`c)!(1 1 3)
q)dic[`b]
1
```

Figure 13: Dictionary example.

Strings and Symbols*

Strings
In q, strings are atomic character lists; therefore, a string list is nested.

Symbols
Symbols are an enumeration type: q internally encodes them with integers but displays symbols like strings with a backtick prefix.

(a) Symbols perform better with assignments and searches.
(b) Symbols are expensive to modify.

One converts strings to symbols with the ‘$ operator and symbols to strings with the “string” function.

```
q)a: "Hello World!"
q)a
"Hello World!"
q)b:`$a
q)b
`Hello World!
q)string b
"Hello World!"
```

Figure 14: String example.

Functions and Loops

Q implements loops by wrapping an operation in a function and iterating the function over a list. This pattern is like R's lapply and python's map.

```
/define a function that displays its arg and returns arg+1
f:{[arg]
  show arg;
  :arg+1; / colon acts as the return operator in this
    context
  };

/test on an atom
f[0]

/loop over a list
f each 0 1 2

/common iteration pattern
f each til 3
```

Function Projections

A common pattern in q is *projection*, where one projects a function with multiple arguments down to a single argument by *freezing* other arguments.

```
\the following function adds two numbers together
g:{[a;b]
  :a+b;
  };

\f is a projection of g, with the first argument fixed to 1
f: g[1;];

\we can now loop over the second argument
f each til 3

\one can loop g without defining the projection f
g[1; ] each til 3
```

Examples

Computing Daily Market Stats

```
\computing daily stats on disk. 0f^x replaces NAs in vector
    x with 0s.
stats: select vol: sdev 0f^(neg 1) + mid% prev mid,
              imbalance: sum 0f^trade*mid,
              adv: sum 0f^abs trade*mid
              by date, stock from bin10;

\avoid look-ahead bias by shifting data. A trading month has
     roughly 20 days.
stats: update vol: prev mavg[20; vol],
              imbalance: prev mavg[20; imbalance],
              adv: prev mavg[20; adv]
              by stock from stats;

\loading a month of intraday trading data
tbl: select from bin10 where date within (2019.01.01
    2019.01.31);

\joining the daily stats with the trading data
tbl: tbl lj `date`stock xkey update stats;
```

Computing Cumulative Sums and Moving Averages

```
\many signals are moving averages of past trades. It helps
    to normalize variables using adv!
tbl: update cumulativeImb: sums trade%adv,
            imb1min: ema[1%6; trade%adv],
            imb5min: ema[1%30; trade%adv],
            imb30min: ema[1%180; trade%adv],
            imb60min: ema[1%360; trade%adv]
            by date, stock from tbl;

\note the key use of update to parallelize computations by
    date and stock!
\to enable parallel computing, you must specify the number
    of cores when you start q. For instance, to enable
    parallelization over eight cores, use
q.exe -s 8
```

Computing Returns

Why different return horizons?
Different trading strategies operate on different timescales.

* E.g. a one minute strategy focuses on one minute alpha forecasts and price impact: the first minute of the day may require a different model from the middle of the day.
* E.g. a two hour forecast will be less sensitive to the "time of the day". Market news may matter more.

Return Horizons (1/3)

Bad approach to generate returns
Computing variables as aggregation functions over the horizon: this loops over the data multiple times!

time	return
09:30	10bps
09:35	-5bps
09:40	3bps
09:45	8bps
09:50	-1bps
09:55	-8bps
10:00	-13bps
10:05	2bps

10min returns

time	return
09:30	10bps
09:35	-5bps
09:40	3bps
09:45	8bps
09:50	-1bps
09:55	-8bps
10:00	-13bps
10:05	2bps

15min returns

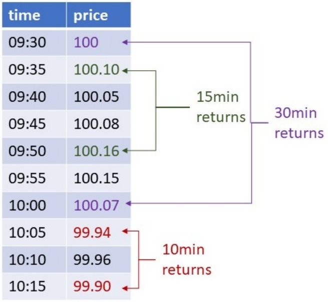
time	return
09:30	10bps
09:35	-5bps
09:40	3bps
09:45	8bps
09:50	-1bps
09:55	-8bps
10:00	-13bps
10:05	2bps

30min returns

Good approach to generate returns

(a) Use cumulative variables, e.g., prices, impact, cumulative volumes.

(b) For each horizon, compute horizon-specific difference variables.



time	price
09:30	100
09:35	100.10
09:40	100.05
09:45	100.08
09:50	100.16
09:55	100.15
10:00	100.07
10:05	99.94
10:10	99.96
10:15	99.90

Simple shift in Python Pandas

```
df['ret10min'] = df['price'].shift(2) % df['price'] - 1
df['ret30min'] = df['price'].shift(6) % df['price'] - 1
```

instead of expensive aggregations

```
df['ret10min'] = df['ret'].groupby('10min').aggregate('cumulateRet')
df['ret30min'] = df['ret'].groupby('30min').aggregate('cumulateRet')
```

Return Horizons (3/3)

```
\optional step if you aren't sure the grid is sorted by time
    . Sorting by date, stock accelerates the grouping.
tbl: `date`stock`time xasc tbl;

\xprev[n; ] shifts data by n steps. Negative n look into the
     future.

\For grids, we look ahead a fixed number of steps on the
    grid within a given (date, stock) group. This assumes
    the grid is sorted by time!
tbl: update retEod: (neg 1) + last[mid]%mid,
            ret1min: (neg 1) + xprev[-6; mid]%mid,
            ret5min: (neg 1) + xprev[-30; mid]%mid,
            ret30min: (neg 1) + xprev[-180; mid]%mid,
            ret60min: (neg 1) + xprev[-360; mid]%mid,
            by date, stock from tbl;
```

Computing Returns for Non-grids (e.g. Intervals)

```
tbl: select startTime: time, mid from events;

\ Step (a): duplicate rows for each horizon to define
    interval end times
tbl: tbl cross ([]horizon: 00:01:00 00:05:00 00:30:00
    01:00:00);
tbl: update endTime: startTime + horizon from tbl;

\ Step (b): Apply an as-of joins to endTimes
tmpTbl: select endTime: time, endMid: mid from events;
tbl: aj[`endTime; tbl; tmpTbl];

\ Step (c) Finalize return computation using the start and
    end states
outputTbl: update ret: (neg 1) + endMid%mid from intervalTbl
    ;
```

Weekly Summary

Learn qSql queries.
It's a useful data grammar: make sure you are comfortable solving standard data manipulation problems with queries.

Keep kdb+'s idiosyncracies in mind.
The right to left evaluation is an unfortunate reality. The earlier you get used to it, the better.

Take advantage of kdb+'s speed.
If an operation feels slow, you are likely doing it "wrong".

Document your code.
Documenting code is a good practice. It's particularly important for Q, which is a *terse* language.

Questions?

Next week
First module: using price impact models in trading algorithms.

(a) Mathematical foundation of price impact.

(b) Example on the Obizhaeva and Wang (OW) model.

(c) Translate alpha signals into trades.