it's about time

Developer Brief

Order Book: a kdb+ Intra-Day Storage and Access Methodology

Author:

Niall Coulter, who joined First Derivatives in 2005, has worked on many kdb+ algorithmic trading systems related to both the equity and FX markets. Based in New York, Niall is a Technical Architect for First Derivatives' Kx products, a suite of high performance data management, event processing and trading platforms.

Table of Contents

1 INTRODUCTION

The purpose of this whitepaper is to describe some of the structures available for storing a real-time view of an order book in kdb+ and to provide some examples of how to effectively query each of these structures.

Efficient maintenance of, and access to, order book data is a valuable component of any program trading, post-trade and compliance application. Due to the high-throughput nature of the updates, which typically arrive on a per symbol basis, an order book has to be maintained and queried on each tick. A vanilla insert of data directly into a quote table may be efficient when updating the data but proves very costly when trying to extract the required order ladders. Alternatively, it would be possible to store the data in an easy-to-query manner but this would increase the overhead and latency to each update, potentially leading to a bottleneck and backlog of messages.

This whitepaper discusses various methods that can be used to solve the aforementioned challenges associated with maintaining a real-time view of an order book. The execution of these methods is first applied to a simplified data set before being applied to a real business use-case. Performance metrics are provided throughout.

2 ORDER BOOK DATA

There are many ways to receive order book data and, within each of them, many different schemas can be used to store the information. For instance, order book updates can range from 'add', 'modify' or 'delete' type messages with a unique order identifier, to something as simple as receiving total available quantity at each price level. The examples below focus on the latter case for simplicity but the same principles can be applied to the more complex individual orders.

We will use a very basic order book schema and make the assumption that we deal with one symbol at a time when we receive an update.

```
/simple example order book schema
q)book:([]time:`second$();sym:`$();side:`char$();price:`float$();size:`int$())

/create sample data
q)s:(),`FDP
q)px:(),5
q)create:{[n]dict:s!px;sym:n?s;side:n?"BS";
          price:((+;-)side="B") .'flip(dict sym;.05*n?1+til 10);
          sample:flip`time`sym`side`price`size!
            (09:30:00+til n;sym;side;price;100*n?1+til 10)}
q)x:create 10
```

2.1 Different kdb+ Structures to Store Order Book Data

We examine four possible structures that could be used to store a real-time view of the order book data described above. While not a definitive list, it is useful to demonstrate a logical progression in the different columns that can be used to key this type of data.

```
/structure 1 - table keyed on sym,side,price
q)book3key:`sym`side`price xkey book

/structure 2 - separate tables keyed on sym,price
q)bidbook2key:askbook2key:`sym`price xkey book

/structure 3 - table keyed on side,price in dictionary keyed on sym
q)bookbysym:(1#`)!enlist`side`price xkey book

/structure 4 - separate tables keyed on price in dictionary keyed on sym
q)bidbookbysym:askbookbysym:(1#`)!enlist`price xkey book
```

It is important to consider potential issues that may be encountered when using a column of type float as one of the keys in our structures. Due to precision issues that can occur when using floats (which have been discussed enough across various forums as not to warrant repetition here), it is possible that you may discover what appears to be a duplicate keyed row in your table. Mindful that the example below is contrived, imagine receiving the zero quantity update to a price level and the following occurs:

```
q)bookfloat:`sym`side`price xkey book
q)`bookfloat upsert flip`time`sym`side`price`size!
      (09:30:00 09:30:01;`FDP`FDP;"BB";4.950000001 4.949999996;100 0)
`bookfloat
q)bookfloat
```

sym	side	price	time	size
FDP	B	4.95	09:30	100
FDP	B	4.95	09:30	0

```
/display the maximum precision possible
q)\P 0
q)bookfloat
```

sym	side	price	time	size
FDP	B	4.9500000010000003	09:30	100
FDP	B	4.9499999959999998	09:30	0

This could lead you to believe a price is still available when it is in fact not, resulting in an incorrect view of the order book.

An immediate solution would be to consider using integers for price values and utilize a price multiplier on a by-symbol basis. The integer conversion could either be done at the feedhandler level or inside the process maintaining the order book state.

```
q)pxm:(0#`)!0#0N
q)pxm[`FDP]:100
q)pxmf:{`int$y*100^pxm x}
q)pxmf . x`sym`price
495 450 470 490 530 505
```

Using a column of type float as one of the keys in our structures is a point for consideration when deciding on a schema to maintain book state. Floating point issues aside, in the interest of consistency and ease of reading, the remainder of this whitepaper will continue to use price as a float.

2.2 Maintaining the Order Book Intra-Day

Now let us consider each of the functions required to upsert the data into each of our four structures. We will assume upon data receipt that the standard upd function is invoked with t and x parameters; t is the table name (not used in the functions below) and x is a table containing the data. As previously specified, we will assume we only ever receive one symbol in each upd callback and, initially at least, we can receive data for both sides of the book.

```
/sample upd functions to populate each structure
q)updSimple:{[t;x]`book3key upsert x}

q)updBySide:{[t;x]
        if[count bid:select from x where side="B";`bidbook2key upsert bid];
        if[count ask:select from x where side="S";`askbook2key upsert ask];
        }

q)updBySym:{[t;x]s:first x`sym;
        bookbysym[s],:x;
        }

q)updBySymSide:{[t;x]s:first x`sym;
        if[count bid:select from x where side="B";bidbookbysym[s],:bid];
        if[count ask:select from x where side="S";askbookbysym[s],:ask];
        }
```

Table 1 – Milliseconds taken to execute each update function 10,000 times

Function	Run 1	Run 2	Run 3	Run 4	Average
do[10000;updSimple[`tablename;x]]	43	41	41	42	41.75
do[10000;updBySide[`tablename;x]]	92	92	93	91	92
do[10000;updBySym[`tablename;x]]	42	43	42	42	42.25
do[10000;updBySymSide[`tablename;x]]	79	80	81	80	80

In Table 1, we can clearly see it is faster to do one upsert into a single table rather than two upserts into two tables, with the double upsert functions taking almost twice as long as the single.

If we change x to contain only one side and rerun our two slower functions again, we start to approach similar timings to the single upsert functions, as shown in Table 2 overleaf.

```
q)x:select from x where side="B"
```

Table 2 - Milliseconds taken to execute each update function 10,000 times if single-sided

Function	Run 1	Run 2	Run 3	Run 4	Average
do[10000;updBySide[`tablename;x]]	60	59	59	60	59.5
do[10000;updBySymSide[`tablename;x]]	54	54	53	53	53.5

Taking this further, assuming we only ever receive updates for a single side, we could alter our upd function definitions for further efficiencies as depicted in Table 3.

```
q)updBySide2:{[t;x]
        $["B"=first x`side;`bidbook2key;`askbook2key]upsert x;}

q)updBySymSide2:{[t;x]s:firstx`sym;
        $["B"=first x`side;bidbookbysym[s],:x;askbookbysym[s],:x];}
```

Table 3 - Milliseconds taken to execute each modified update function 10,000 times if single-sided

Function	Run 1	Run 2	Run 3	Run 4	Average
do[10000;updBySide2[`tablename;x]]	35	34	35	35	34.75
do[10000;updBySymSide2[`tablename;x]]	31	30	29	29	29.75

3 ACCESSING THE ORDER BOOK

Now that the data is being maintained in each of our four structures, it is worthwhile considering some common queries that we might wish to perform, the most obvious being extracting top-of-book. The following functions all take a single symbol as an argument and return the same result; a dictionary of bid and ask e.g. `bid`ask!4.95 5.1

For the sake of brevity, the code below does not exclude price levels where the size is equal to zero. However, handling this is straight forward and should not have a significant impact on performance.

```
/sample functions to extract top-of-book
q)getTopOfBook:{[s]
        b:exec bid:max price from book3key where sym=s,side="B";
        a:exec ask:min price from book3key where sym=s,side="S";
        b,a}

q)getTopOfBookBySide:{[s]
        b:exec bid:max price from bidbook2key where sym=s;
        a:exec ask:min price from askbook2key where sym=s;
        b,a}

q)getTopOfBookBySym:{[s]
        b:exec bid:max price from bookbysym[s]where side="B";
        a:exec ask:min price from bookbysym[s]where side="S";
        b,a}

q)getTopOfBookBySymSide:{[s]
        b:exec bid:max price from bidbookbysym s;
        a:exec ask:min price from askbookbysym s;
        b,a}
```

The times taken to execute each of the functions are illustrated below:

Table 4 - Milliseconds taken to execute each top-of-book function 10,000 times

Function	Run 1	Run 2	Run 3	Run 4	Average
do[10000;getTopOfBook`FDP]	32	32	32	33	32.25
do[10000;getTopOfBookBySide`FDP]	21	21	21	22	21.25
do[10000;getTopOfBookBySym`FDP]	24	23	24	23	23.5
do[10000;getTopOfBookBySymSide`FDP]	17	17	17	17	17

Table 4 results clearly demonstrate the getTopOfBookBySymSide function takes the least amount of time to return top-of-book, albeit by a very small fraction of a microsecond. However, with some small refinements we can achieve even greater improvements.

```
q)getTopOfBookBySymSide2:{[s]
      `bid`ask!(max key[bidbookbysym s]`price;min key[askbookbysym s]`price)}
```

Table 5 - Milliseconds taken to execute the modified top-of-book function 10,000 times

Function	Run 1	Run 2	Run 3	Run 4	Average
do[10000;getTopOfBookBySymSide2`FDP]	9	9	9	10	9.25

Table 5 shows that the getTopOfBookBySymSide2 function is over three times faster than the getTopOfBook function using the book3key table and approximately twice as fast as the getTopOfBookBySide and getTopOfBookBySym functions using the bid/askbook2key and bookbysym structures respectively. This represents a significant saving if top-of-book is being calculated on every update which could help to alleviate back pressure on the real-time book process and increase throughput of messages.

3.1 Calculating the Top Two Levels

Another common query is to extract the top two levels of the book from each side. Again, the following functions all take a single symbol as an argument and return the same result; a dictionary of bid1, bid, ask and ask1 e.g. `bid1`bid`ask`ask1!4.9 4.95 5.1 5.15

```
/sample functions to extract top 2 levels
q)getTop2Book:{[s]
        b:`bid`bid1!2 sublist desc exec price from book3key where sym=s,side="B";
        a:`ask`ask1!2 sublist asc exec price from book3key where sym=s,side="S";
        reverse[b],a}
q)getTop2BookBySide:{[s]
        b:`bid`bid1!2 sublist desc exec price from bidbook2key where sym=s;
        a:`ask`ask1!2 sublist asc exec price from askbook2key where sym=s;
        reverse[b],a}
q)getTop2BookBySym:{[s]
        b:`bid`bid1!2 sublist desc exec price from bookbysym[s]where side="B";
        a:`ask`ask1!2 sublist asc exec price from bookbysym[s]where side="S";
        reverse[b],a}
q)getTop2BookBySymSide:{[s]
        b:`bid`bid1!2 sublist desc exec price from bidbookbysym s;
        a:`ask`ask1!2 sublist asc exec price from askbookbysym s;
        reverse[b],a}
```

Table 6 - Milliseconds taken to execute each top 2 levels of book function 10,000 times

Function	Run 1	Run 2	Run 3	Run 4	Average
do[10000;getTop2Book`FDP]	75	74	75	74	74.5
do[10000;getTop2BookBySide`FDP]	63	62	62	61	62
do[10000;getTop2BookBySym`FDP]	64	64	63	64	63.75
do[10000;getTop2BookBySymSide`FDP]	58	58	58	58	58

Once again, Table 6 results show the getTop2BookBySymSide function returning the top two levels of the book in the least amount of time. As was the case in the last example, we can further optimize the function to achieve greater improvement.

```
q)getTop2BookBySymSide2:{[s]
        bid:max bids:key[bidbookbysym s]`price;
        b:`bid1`bid!(max bids where not bids=bid;bid);
        ask:min asks:key[askbookbysym s]`price;
        a:`ask`ask1!(ask;min asks where not asks=ask);
        b,a}
```

Table 7 - Milliseconds taken to execute the modified top 2 levels of book function 10,000 times

Function	Run 1	Run 2	Run 3	Run 4	Average
do[10000;getTop2BookBySymSide2`FDP]	28	28	28	28	28

By using min and max in the getTop2BookBySymSide2 function instead of asc and desc, approximately half the time is needed by the other four functions. Again, this could help to alleviate back pressure on the real-time book process and increase throughput of messages.

3.2 Business Use Case

Now let us consider more realistic examples by increasing the number of symbols across a range of one thousand to five thousand, in one thousand increments, and adding twenty levels of depth on each side. For these examples, it makes sense to apply some attributes to the different structures to maximize query efficiency. We will not profile the update functions again since we used the assumption of only ever receiving updates for a single symbol.

```
/structure 1 - apply g# to sym and side columns
q)book3key:update`g#sym,`g#side from`sym`side`price xkey book

/structure 2 - apply g# to sym column
q)bidbook2key:askbook2key:update`g#sym from`sym`price xkey book

/structure 3 - apply u# to dictionary key and g# to side column
q)bookbysym:(`u#1#`)!enlist update`g#side from`side`price xkey book

/structure 4 - apply u# to dictionary key
q)bidbookbysym:askbookbysym:(`u#1#`)!enlist`price xkey book

q)s,:upper (neg ns:1000)?`5;
q)px,:1+til ns;
q)createBig:{[n]dict:s!px;sym:n?s;side:n?"BS";
     price:((+;-)side="B") .'flip(dict sym;.05*n?1+til 20);
     sample:flip`time`sym`side`price`size!
          (asc n?09:30:00+til 23400;sym;side;price;100*n?1+til 10)}

/applying p# to sym column of x to speed up selection by sym later on
q)x:@[`sym xasc createBig 1000000;`sym;`p#];

/populate the tables remembering we have to split by sym for the last 2 functions
q)updSimple[`tablename;x];
q)updBySide[`tablename;x];
q)updBySym[`tablename]each{[s]select from x where sym=s}each distinct x`sym;
q)updBySymSide[`tablename]each{[s]select from x where sym=s}each distinct x`sym;
```

Table 8 – A table summarizing the millisecond times required to execute each of the functions 10,000 times across a range of symbol counts

Function	1000 Syms	2000 Syms	3000 Syms	4000 Syms	5000 Syms
do[10000;getTopOfBook`FDP]	36	36	36	36.5	36.5
do[10000;getTopOfBookBySide`FDP]	21	21.5	21.5	21.5	21.5
do[10000;getTopOfBookBySym`FDP]	23	23	23.5	23.5	24
do[10000;getTopOfBookBySymSide`FDP]	19	19	19	19	19
do[10000;getTopOfBookBySymSide2`FDP]	11	11	11	11	11
do[10000;getTop2Book`FDP]	76	76.5	76.5	76.5	76.5
do[10000;getTop2BookBySide`FDP]	61	62	62	62	62
do[10000;getTop2BookBySym`FDP]	64	64	64	64	64
do[10000;getTop2BookBySymSide`FDP]	60	60	60	60	60
do[10000;getTop2BookBySymSide2`FDP]	34	34	34	34	34

We see that the performance of the above functions remain stable as the number of tickers increase.

4 CONCLUSION

This whitepaper is a brief introduction to some of the options that are available for storing book information and, as mentioned previously, this paper intentionally excludes handling some cases to keep the code as simple and easy to follow as possible (e.g. the getTop… functions do not exclude prices where size is equal to zero and the getTop2… functions have no handling for the case when there are not at least two levels of depth). However, it is straightforward to implement solutions for these cases without having a significant impact on the performance.

Ultimately, the decision as to how to store the data will depend on the two factors identified in this paper:

(1) how the data is received by the process maintaining book state (e.g. single or double sided).
(2) how the data is to be accessed inside the process (e.g. full book or just top n levels required).

How data is received and accessed can vary depending on each individual use case. It is therefore recommended that you experiment with each of the suggestions above, and of course any of the other alternatives that are available, to determine which is best for your particular needs.

All tests performed with kdb+ 2.8 (2012.03.21).