it's about time

Developer Brief

kdb+tick Profiling for Throughput Optimization

Author:

Ian Kilpatrick, who joined First Derivatives in 2001, has worked on several kdb+ systems. Based in Belfast, Ian 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

kdb+ is seen as the technology of choice for many of the world's top financial institutions when implementing a tick capture system. kdb+ is capable of processing large amounts of data in a very short space of time, making it the ideal technology for dealing with the ever-increasing volumes of financial tick data. The core of a kdb+ tick capture system is the tickerplant. Kx have made available the source code for their product kdb+tick which will form the basis of this paper. The purpose of this whitepaper is to discuss factors which influence messaging and throughput performance for a kdb+ based tick capture system and to present a methodology with which this performance can be profiled to assist in optimizing the tick system configuration.

Some of the possible factors are:

- Number of rows in each update
- Size of the data in bytes
- Tickerplant publish frequency
- Number of subscribers

- Network latency and bandwidth
- Disk write speed
- TCP/IP tuning
- Version of kdb+

This paper examines the first 4 of these. All tests were performed on 64 bit Linux with 8 CPUs, using kdb+ version 3.1 (2014.02.08).

For more information on kdb+ tickerplants, see http://code.kx.com/wiki/Startingkdbplus/tick.

NOTE: The results presented in this whitepaper are for indication purposes only. Results will vary for each individual kdb+ system, especially when hardware specifications are taken into consideration.

2 SETUP

We will run a feed simulator which will publish trade data to a tickerplant (TP) on a timer. The trade table has the following schema:

```
trade:([]time:"P"$();sym:`g#"S"$();price:"F"$();size:"I"$();cond:())
```

We will also run an RDB (real-time database) which subscribes to the tickerplant for trade messages and inserts into an in-memory table. All the processes will run on the same server and we will use `taskset` to tie each process to a different CPU.

e.g.

```
taskset -c 0 q tp.q
taskset -c 1 q rdb.q
taskset -c 2 q feedsim.q
```

2.1 Feed simulator code

```
/ connect to tickerplant
h:hopen 8099

/ number of extra columns to add for test 3.2
ex:0

/ number of rows to send in each update
r:10

/ number of updates to send per millisecond
u:1

/ timer frequency
t:1

/ timer function, sends data to the tickerplant
.z.ts:{
  data:(r#.z.p;r?`3;100*r?1.0;10*r?100;r#" ");
  if[ex>0; data,:ex#enlist r#1f];
  if[r=1;data:first each data];
  do[u;neg[h](`upd;`trade;data);neg[h][]];
 }

system"t ",string t

/ stop sending data if connection to tickerplant is lost
.z.pc:{if[x=h; system"t 0"];}
```

2.2 Tickerplant code

```
/ listen on port 8099
\p 8099

/ dictionary to contain handles to publish to
subs:enlist[`trade]!();

/ function to subscribe to a table
sub:{[t] subs[t],:neg .z.w;}

/ remove subscriber if connection is lost
.z.pc:{subs::subs except \: neg x;}

/ create empty log file
logFile:`$":sym",string .z.D;
logFile set ();
numMsgs:0;
fileHandle:hopen logFile;

/ write data to logFile and publish to subscribers, called by the feed sim
upd:{[t;x]
  tm1:.z.p;
  fileHandle@enlist(`upd;t;x);
  numMsgs+:1;
  tplog,:0.001*.z.p-tm1;
  tm2:.z.p;
  subs[t]@\:(`upd;t;x;tm2);
  tppub,:0.001*.z.p-tm2;
 }
```

The tickerplant described above differs from a standard tickerplant in a number of ways. A standard tickerplant would:

- Check to see if the log file already exists on startup and then read the number of messages it contains
- The `sub` function could be enhanced to handle subscribing for certain syms only
- The `upd` function could add a time to the data if it is not present
- The `upd` function would only send `(`upd;t;x)` to the subscribers
- At EOD it would send a message to any subscribers and roll the log file

In the tickerplant described above, we are capturing some extra timing metrics which will allow us to profile the messaging and throughput statistics for the tickerplant. These are described in section 3.

2.3 RDB code

```
/ define trade table
trade:([]time:"P"$();sym:`g#"S"$();price:"F"$();size:"I"$();cond:())

/ number of extra columns to add for test 3.2
ex:0
if[ex>0; trade[`$"col",/:string til ex]:ex#enlist"F"$()]

/ define upd to insert to the table, called by tickerplant
upd:{[t;x;tm2]
  tm3:.z.p;
  insert[t;x];
  tm4:.z.p;
  rdbrecv,:0.001*tm3-tm2;
  rdbupd,:0.001*tm4-tm3;
 }

/ connect to tickerplant and subscribe for trades
h:hopen 8099;
h(`sub;`trade);
```

Again, the RDB described here differs from a standard RDB. A standard RDB would:

- Replay the tickerplant log file on startup to get trades from earlier in the day
- `upd` would only take 2 parameters, the table name and data
- There would be an EOD function defined which would write intra-day tables to disk and then empty the tables

Similarly to the TP code, the code for the RDB records some timing metrics to measure throughput on the RDB. These are described in section 3.

3 TESTS

For each test we will vary certain parameters in the feed simulator code and record the median values for `tplog, tppub, rdbrecv, rdbupd` in the code above where

- `tplog` = median time in microseconds for the tickerplant to write the data to the log file
- `tppub` = median time in microseconds for the tickerplant to publish the data to the subscribers
- `rdbrecv` = median time in microseconds for the RDB to receive the message from the tickerplant
- `rdbupd` = median time in microseconds for the RDB to insert the data

We will also record

- Rows per upd = number of rows received by the tickerplant per update
- Rows per sec = number of rows received by the tickerplant per second
- TP CPU = CPU usage of the tickerplant seen using `top` (with the timing code in the `upd` function removed)
- RDB CPU = CPU usage of the RDB seen using `top` (with the timing code in the `upd` function removed)

Testing Specifications

- All tests were run on 64bit Linux with 8 CPUs, using kdb+3.1 2014.02.08.
- The tickerplant is writing to local disk and the write speed is 400MB/s.
- All the processes run on the same server and are run using `taskset`. Note that kdb+ does not implement IPC compression when publishing to localhost.
- Each test is run until the median times stop changing, which takes roughly 1 or 2 minutes.

3.1 Number of rows in each update

In this test we will vary the number of rows sent in each update by changing the values of `r` (number of rows per update), `u` (number of updates per timer frequency) and `t` (timer frequency in milliseconds) in the feed simulator code. The number of rows received by the tickerplant per second is calculated as:

Rows per second = `r * u * 1000 / t`

Rows per upd (r)	Upds per timer (u)	Timer freq (t)	Rows per sec	tplog	tppub	TP CPU	rdbrecv	rdbupd	RDB CPU
1	10	1	10,000	14	3	31	71	4	12
10	1	1	10,000	19	4	6	80	10	2
100	1	10	10,000	35	7	1	106	46	1
1	30	1	30,000	13	3	92	80	4	24
10	3	1	30,000	16	4	4	85	7	3
100	3	10	30,000	30	6	1	99	44	1
10	10	1	100,000	15	4	32	82	7	17
100	1	1	100,000	32	6	6	103	46	4
1000	1	10	100,000	121	23	2	224	378	3
100	5	1	500,000	28	6	32	105	42	22

Table 1 – Microseconds taken and percent CPU by rows per update

As seen in Table 1, the results mainly depend on the rows per upd (`r`) parameter and not the other 2 parameters. When publishing the data in single rows (rows per upd = 1) we can only achieve about 30,000 rows per second before the TP CPU usage approaches 100%. If the data is in 10 rows per update we can handle over 100,000 rows per second.

Significant time is saved when processing 10 rows at a time. In fact, it takes only a little more time than processing 1 row. This goes to the heart of q: in a vector based language bulk operations are more efficient than multiple single operations.

Rows per upd (r)	1	10	100
tplog	13	17	31
tppub	3	4	6
rdbrecv	75	90	103
rdbupd	4	8	44

Table 2 – Microseconds taken by rows per update

From these results we can conclude that feeds should read as many messages off the socket as possible and send bulk updates to the tickerplant if possible.

3.2 Size of row in bytes

In this test we will vary the number of columns in the trade table by changing `ex` (number of extra columns) in the feedsim and RDB code.

- The times for ex=0 have already been found in section 3.1.
- For each update message, we determine the size using `count -8!x`.

Num extra cols (ex)	Rows per upd (r)	Rows per sec	tplog	tppub	TP CPU	rdbrecv	rdbupd	RDB CPU	Size of upd in bytes
0	1	10,000	14	3	31	82	4	12	73
10	1	10,000	14	4	34	87	6	16	213
50	1	10,000	17	5	43	79	12	38	773
0	10	100,000	15	4	32	86	7	17	334
10	10	100,000	18	5	40	78	8	19	1194
50	10	100,000	33	7	60	92	14	40	4634
0	100	100,000	28	6	6	106	41	4	2944
10	100	100,000	50	8	13	187	46	8	11004
50	100	100,000	141	16	28	336	61	23	43244

Table 3 – Microseconds taken and percent CPU by size of row

We can see the `tplog`, `tppub` and `rdbupd` times increase as the number of columns increase, as we would expect. The increase is more noticeable for large bulk updates: if there is only 1 row per update then adding 10 columns to the published data only increases the CPU usage of the tickerplant by 10%.

3.3 Publish frequency

In this test we will alter the tickerplant behaviour so that it publishes to subscribers on a timer. Specifically, we will examine the following three scenarios:

- The tickerplant writes each update to disk individually and publishes each update to subscribers as soon as it is received (standard zero-latency tickerplant)
- The tickerplant writes each update to disk individually but publishes to subscribers on a timer
- The tickerplant both writes to disk and publishes to subscribers on a timer

By batching up updates we can reduce the load on the tickerplant and the RDB. However, this will also result in a delay to RDB receiving the data. Of course, publishing on a timer is not suitable if the subscribers need the data immediately.

We need to make the following adjustments to the tickerplant code in order to publish on a timer:

```
/ define trade table
trade:([]time:"P"$();sym:`g#"S"$();price:"F"$();size:"I"$();cond:())

/ write to disk and insert to the local table
upd:{[t;x]
  tm:.z.p;
  fileHandle@enlist(`upd;t;x);
  numMsgs+:1;
  insert[t;x];
  tpupd,:0.001*.z.p-tm;
 }

/ publish the data in the local table and clear
.z.ts:{
  tm:.z.p;
  {[t]
    if[0=count value t; :()];
    subs[t]@\:(`upd;t;value t);
    .[t;();0#];
  } each enlist`trade;
  tpflush,:0.001*.z.p-tm;
 }

/ run .z.ts every 100 milliseconds
\t 100
```

To further reduce the load we will buffer the messages and only write to the on-disk log file on a timer as well. However, it should be noted that in the event of a tickerplant going down, more data will be lost if the data is being logged on a timer than if the data is being written on every update.

In order to write updates to disk on a timer, we need to make the following changes to the tickerplant code:

```
/ define trade table
trade:([]time:"P"$();sym:`g#"S"$();price:"F"$();size:"I"$();cond:())

/ insert to the local table
upd:{[t;x]
  tm:.z.p;
  insert[t;x];
  tpupd,:0.001*.z.p-tm;
 }

/ publish the data in the local table, write to disk and clear
.z.ts:{
  tm:.z.p;
  {[t]
    if[0=count value t; :()];
    subs[t]@\:(`upd;t;value t);
    fileHandle@(`upd;t;value t);
    numMsgs+:1;
    .[t;();0#];
  } each enlist`trade;
  tpflush,:0.001*.z.p-tm;
 }

/ run .z.ts every 100 milliseconds
\t 100
```

Rows per upd	Rows per sec	Timer freq	Pub on timer	Write on timer	tpupd	tpflush	TP CPU	rdbupd	RDB CPU
1	10,000	0	N	N	13	0	31	3	12
1	10,000	100	Y	N	13	36	22	258	0.1
1	10,000	100	Y	Y	3	169	9	273	0.1

Table 4 – Microseconds taken and percent CPU by publish frequency

Where

- Timer freq = frequency the timer in the tickerplant is run in milliseconds
- `tpupd` = median time in microseconds to run `upd` in the tickerplant
- `tpflush` = median time in microseconds to run the timer (`.z.ts`) in the tickerplant

In Table 4 above we can see that when publishing on the timer, the tickerplant `upd` function still takes roughly the same time as in zero latency mode, but we are only publishing data 10 times a second which reduces the overall load: the TP CPU usage has decreased from 31% to 22%. The RDB CPU usage decreases from 12% to 0.1% as it is only doing 10 bulk updates per second instead of 10,000 single updates per second. Only writing to disk 10 times a second reduces the load on the tickerplant further. The improvements will be greater the more updates the tickerplant receives per second.

3.4 Number of subscribers

In this test, we will examine how the number of processes that are subscribing to a tickerplant's data can affect the throughput of the tickerplant. We will run multiple subscribers/RDBs and see the effect on the tickerplant publish time and RDB receive time. We will collect the following metrics:

- Last `rdbrecv` = median time in microseconds for the last RDB in the tickerplant subscription list (`subs` dictionary) to receive the message from the tickerplant
- First `rdbrecv` = median time in microseconds for the last RDB in the tickerplant subscription list (`subs` dictionary) to receive the message from the tickerplant
- Num subs = number of subscribers

Rows per upd	Upds per timer	Timer freq	Rows per sec	Num subs	tppub	First rdbrecv	Last rdbrecv	TP CPU
1	1	1	1,000	1	3	85	85	3
1	1	1	1,000	3	4	172	178	4
1	1	1	1,000	5	6	148	296	6
1	1	1	1,000	10	10	265	343	10
10	1	1	10,000	1	3	88	88	3
10	1	1	10,000	3	5	175	181	5
10	1	1	10,000	5	6	155	318	7
10	1	1	10,000	10	11	224	540	12
100	1	1	100,000	1	21	97	97	6
100	1	1	100,000	3	58	177	324	10
100	1	1	100,000	5	95	257	330	15
100	1	1	100,000	10	185	449	682	30

Table 5 – Microseconds taken by number of subscribers

We can see that increasing the number of subscribers increases the tickerplant publish time, first RDB receive time and last RDB receive time. The first RDB receive time increases because the data is written to each internal message queue and the queues are not flushed until the tickerplant publish function returns.

If there are multiple subscribers to a tickerplant it might be worth considering a chained tickerplant to reduce the number of subscribers. Only the chained tickerplant and the subscribers which need the data as quickly as possible would subscribe to the main tickerplant. Then other subscribers would subscribe to the chained tickerplant to get the data.

4 CONCLUSION

This whitepaper examined some key factors which can influence the performance and throughput of a kdb+ tickerplant. We established a methodology to profile the performance of a kdb+ tickerplant and used it to focus on four key areas which can affect the tickerplant's throughput:

1. **Number of rows per update**

   Feeds should read as many messages off the socket as possible and send bulk updates to the tickerplant if possible. Bulk updates will greatly increase the maximum throughput achievable, e.g. we found processing a bulk update of 10 messages took only slightly longer then processing an update of 1 message, and hence the CPU usage was nearly 10 times lower (for the same number of messages per second).

2. **Size of each update in bytes**

   Reducing the size of the data can improve throughput as writing to disk and sending the data will be faster. The improvement is more noticeable on bulk updates.

3. **Publish frequency**

   Buffering the messages in the tickerplant and publishing on a timer improves throughput. The CPU usage of the RDB also decreases as the data is being batched up into fewer updates per second. However, it is not suitable if the subscribers need the data immediately. Writing the messages to disk on a timer improves throughput further but more data could be lost if the tickerplant dies. The improvement is more noticeable as the number of updates per second increases.

4. **Number of subscribers**

   Adding more subscribers increases the load on the tickerplant. To reduce this, consider using a chained tickerplant. Even existing subscribers will be affected if more subscribers are added since the internal message queues for each subscriber are all written to before the queues are flushed.

It should be noted that these are not the only factors that contribute to a tickerplant's performance. A complete analysis would also examine the effects that network latency and bandwidth, disk write speed and TCP/IP tuning have on tickerplant performance.

The results shown in this paper are not representative of kdb+ systems as a whole. Results for each individual kdb+ system will vary due to various hardware and software considerations. However, the code and methodology used in this paper could serve as a starting point to any developers wishing to profile and optimize their own systems.

All tests were run using kdb+ version 3.1 (2014.02.08)