q for Gods Whitepaper Series (Edition 20)

Data Recovery for kdb+tick

July 2014

Author:

Fionnbharr Gaston, who joined First Derivatives in 2011, is a kdb+ consultant currently based in Singapore. He has worked with a number of top-tier investment banks and exchanges in the UK, Asia and Australia, working on applications ranging from market data capture to market surveillance.

TABLE OF CONTENTS

1 INTRODUCTION

Kx Systems freely offers a complete tick capture product which allows for the processing, analysis and historical storage of huge volumes of tick data in real time. This product, known as kdb+tick, is extremely powerful, lightweight and forms the core of most kdb+ architectures. The tickerplant lies at the heart of this structure. It is responsible for receiving data from external feedhandlers and publishing to downstream subscribers. Perhaps the most important aspect of the tickerplant is how it logs every single message it receives to a binary log file. In the event of a subscriber process failing, this log file can be used to restore any missing data. However, should this log file become corrupt, or if there are problems during recovery, a complete replay of the log file may not be possible.

This paper will explore the nature of tickerplant log files and their importance in a kdb+tick system. We will also examine how log files can become corrupted and the various solutions that exist for this scenario.

All tests were run using kdb+ version 3.1 (2014.03.27).

2 kdb+tick

kdb+tick is an architecture which allows the capture, processing and querying of data in real-time and historically. It typically consists of:

- **The Feedhandler**

A feedhandler is a process which captures external data and translates it into kdb+ messages. Multiple feed handlers can be used to gather data from a number of different sources and feed it to the kdb+ system for storage and analysis.

- **The Tickerplant**

The tickerplant (TP) is a specialized kdb+ process that operates as a link between the client's data feed and a number of subscribers. It receives data from the feedhandler, appends a time stamp to it and saves it to a log file. It publishes this data to a real-time database (RDB) and any clients which have subscribed to it, and then purges its tables of data. In this way the tickerplant uses very little memory, whilst a full record of intra-day data is maintained in the real-time database.

- **Real-time Database**

The real-time database (RDB) holds all the intra-day data in-memory to allow for fast, powerful queries. At startup, the RDB sends a message to the tickerplant and receives a reply containing the data schema, the location of the log file, and the number of lines to read from the log file. It then receives subsequent updates from the TP as they are published.

- **Historical Database**

The historical database (HDB) consists of on-disk kdb+ data, typically split into date partitions. A q process can read this data and memory-map it, allowing for fast queries across a large volume of data. The RDB is instructed to save its data to the HDB at EOD.

Previous editions of this series have studied this architecture in detail, including Edition 19: *kdb+tick Profiling for Throughput Optimization* (http://www.firstderivatives.com/Products_pdf.asp?downloadflyer=q_for_Gods_May_2014)

This paper will primarily consider the relationship between the TP and RDB and in particular the use of tickerplant logs when recovering lost data in an RDB.

2.1 Schemas

For the worked examples in this paper, we will make use of a simple trade table schema. Later, we shall look at `accounts` - a simple position limits table, which we will key in a real-time engine process. The schemas are defined in the file `sym.q`:

```
trade:([]time:`timespan$();sym:`$();side:`char$();size:`long$();price:`
float$() );
accounts:([] time:`timespan$(); sym:`$(); curr:`$(); action:`$();
limit:`long$());
```

2.2 Tick scripts

kdb+ tick is freely available from code.kx.com and contains a few short, yet powerful scripts:

```
tick.q      # runs a standard tickerplant
tick
 |
  -  r.q    # runs a standard real-time database
  -  u.q    # contains functions for subscription and publication
```

For more details on kdb+tick, please refer to http://code.kx.com/wiki/Startingkdbplus/tick.

2.2.1 kdb+ messages and upd function

A kdb+ message takes the form of a list of `(functionname;tablename;tabledata)`. Here, `functionname` and `tablename` are symbol types. `tabledata` is a row of data to be inserted into `tablename`:

```
`upd `trade    (0D14:56:01.113310000;`AUDUSD;"S";1000;96.96)
`upd `trade    (0D14:56:01.115310000;`SGDUSD;"S";5000;95.45)
`upd `trade    (0D14:56:01.119310000;`AUDUSD;"B";1000;95.08)
`upd `trade    (0D14:56:01.121310000;`AUDUSD;"B";1000;95.65)
`upd `trade    (0D14:56:01.122310000;`SGDUSD;"B";5000;98.14)
```

In a standard tick system, each kdb+ message will call a function named `upd`. Each process may have a different definition of this function. A TP will publish each time the `upd` function is called (if its timer is not set to batch the data), and an RDB will simply insert the data into the relevant table.

3 RECOVERY

3.1 Writing a tplog

Should the TP fail, or be shut down for any period of time, all downstream subscribers will not receive any published data for the period of its downtime. This data typically will not be recoverable – thus it is imperative that the TP remains always running and available.

Every message that the tickerplant receives is written to a q binary file, called the tickerplant log file (tplog). The tickerplant maintains some key variables which are important in the context of data recovery for subscribers:

- `.u.l` – This is a handle to the log file which is created at startup. This is used to write each message to disk.
- `.u.L` – This is the path to the log file. In a standard tickerplant, the name of the log file will be a combination of the first parameter passed to tick.q (the name of the schema file, generally '`sym`') and the current date.
- `.u.i` – This is the total count of messages in the log file.
- `.u.j` – The total count of messages in the tickerplant (`.u.i` plus what is buffered in memory).

```
//start tickerplant
>q tick.q sym . -p 5010
q).u.L
`:./sym2014.05.03
q).u.l
376i
q).u.i
0
```

The `upd` function is called each time a TP receives a message. Within this function, the TP will write the message to the tplog:

```
//from u.q
upd:{[t;x]
...
//if the handle .u.l exists, write (`upd;t;x) to the tplog; increment
.u.j by one
if[l; l enlist (`upd;t;x);j+:1]
```

3.2 Replaying a tplog

Recovery of an RDB typically involves simply restarting the process. On startup, an RDB will subscribe to a TP and will receive information about the message count (`.u.i`) and location of the tplog (`.u.L`) in return. It will then replay this tplog to recover all the data that has passed through the TP up to that point in the day. The replay is achieved using `-11!`, the streaming replay function. This is called within `.u.rep` which is executed when the RDB connects to the TP:

```
// From r.q :
.u.rep:{…;-11!y;…};
```

kdb+ messages were described in section 2.2.1. In a typical RDB, `functionname` will be `` `upd `` which will perform an insert. Therefore, executing a single line in the logfile will be equivalent to `insert[`tablename;tabledata]`:

```
//from r.q
upd:insert;

q)trade
time                 sym    side size price
-----------------------------------------
q) value first get`:logfile
,0
q)trade
time                 sym    side size price
-----------------------------------------
0D14:56:01.113310000 AUDUSD S    1000 96.96
```

3.3 -11! Functionality

-11! Is an internal function that will read each message in a tplog in turn, running the function `functionname` on the (`tablename;tabledata`) parameters .

In this section, we will focus on normal usage of the `-11!` function where the tplog is uncorrupted, and move onto discuss how to use it to recover from a corrupted tplog in section 4.

3.3.1 -11!`:tplog

There are three distinct usages of `-11!` when passed a list of two parameters, depending on the value of the first parameter. Passing only one parameter is functionally equivalent to passing a first parameter of `-1`:

```
-11!(-1;`:tplog)
-11!`:tplog
```

`-11!` will replay the tplog into memory, thus recovering the data. This can be reproduced by running "`value each get`:tplog`" but `-11!` uses far less memory. `-11!` will read and process each row of the tplog in turn, whereas "`get`:tplog`" will read the whole tplog into memory, then "`value each`" will execute each row in turn:

```
q)\ts value each get`:sym2014.05.07
1 4198832
q)\ts -11!`:sym2014.05.07
1 4195344
q)\ts -11!`:sym2014.05.07
1204 4257168j
q)\ts value each get `:sym2014.05.07
1690 296580768j
```

The return value from `-11!`:logfile` is the number of tplog lines executed:

```
q)count trade
0
/ playback the logfile
q)-11!`:sym2014.05.07
18
q)count trade
18
```

3.3.2 -11!(-2; `:tplog)

When the first parameter is -2, there are two possible forms of the output. If the tplog is uncorrupted, the return value is a single long number describing the total number of rows in the logfile. This result is equivalent to "`count get`:tplog`", but `-11!` is faster and more memory efficient:

```
q)\ts -11!`:sym2014.04.30
19502 5276800
q)\ts value each get `:sym2014.04.30
21321 583811984
```

Should a tplog be corrupted in some way, running `-11!(-2;`:tplog)` will return two values: the first describes the number of good chunks in the tplog while the second describes the number of replayable bytes i.e. the point in the log file where the replaying function hits an error:

```
q)-11!(-2;`:sym2013.10.29)
46333621
46756601608j
```

This means that there are 46,333,621 valid chunks in the tplog, and 467,56,601,608 valid bytes. Anything after this point is considered corrupt and not replayable.

3.3.3 -11!(n;`:tplog)

This performs streaming execution of the first `n` chunks of the specified tplog. This is helpful when the position of the corrupt message is known (determined by executing `-11!(-2;`:tplog`)), The return value is the number of tplog chunks executed:

```
q)-11!(46333621; `:sym2013.10.29)
46333621
```

4 REPLAY ERRORS

4.1 Corrupt tplog

It is possible that a tplog may become corrupted. For example, the tickerplant process could die mid-write (e.g. due to hardware failure). For recovery, we wish to isolate the valid parts of the tplog, discarding the corrupted sections. This can be done with a combination of the various forms of `-11!`.

First, we attempt to replay a corrupted tplog:

```
q)-11!(`:sym2013.10.29)
'badtail
```

`'badtail` is a q error that indicates an incomplete transaction at end of the file.

Next, we identify the position of the corruption in the tplog using `-11!(-2;x)`:

```
q)-11!(-2;`:sym2013.10.29)
46333621
46756601608j
```

There are 46,333,621 valid chunks in the tplog, and the valid portion of the log file is 46,756,601,608 bytes in size.

For safety, we will back up the corrupt tplog. We will also remove user write permission to reduce the risk of accidentally affecting this file:

```
$ mv sym2013.10.29 sym2013.10.29_old
$ chmod u-x sym2013.10.29_old
```

Within a new q session, we create variables to point to this tplog, and create a handle to a new tplog. We assume the current working directory of this q session is the same as where the tplog is located:

```
q) old:`:sym2013.10.29_old
```

Initiate a new, empty file, and open a handle to it:

```
q)new:`:sym2013.10.29_new
q)new set ()
`:sym2013.10.29_new
q)h:hopen new
```

We define the `upd` function within this new q process to write to this handle:

```
q)upd:{[t;x]h enlist(`upd;t;x)}
```

We use `-11!` to replay the first 46,333,621 chunks of the corrupt tplog. This will omit the last, bad message:

```
q)-11!(46333621;old)
46333621
```

The `upd` function (as defined in this process) is being called in each one of these messages. This will write each one of these messages in turn to the handle h, which is streaming the message to the new log file. The equivalent execution is:

```
q)enlist first get`:sym2014.05.07
q)get`:sym2014.05.03
`upd `trade (0D14:56:01.113310000;`AUDUSD;"S";1000;96.96)
q)h enlist first get`:sym2014.05.07
392i
// the return value is the value of the handle
q)get new
`upd `trade (0D14:56:01.113310000;`AUDUSD;"S";1000;96.96)
```

This will give us a new tplog with the corrupt parts removed. This can now be replayed to restore the data in the RDB up until the point of corruption:

```
q)trade
time sym side size price
------------------------

//recover new tplog
q) -11! new
46333621j

q)trade
time                  sym    side size price
-----------------------------------------
0D14:56:01.113310000 AUDUSD S    1000 96.96
0D14:56:01.115310000 SGDUSD S    5000 95.45
0D14:56:01.119310000 AUDUSD B    1000 95.08
0D14:56:01.121310000 AUDUSD B    1000 95.65
0D14:56:01.122310000 SGDUSD B    5000 98.14
0D14:56:01.124310000 AUDUSD S    1000 97.17
0D14:56:01.126310000 AUDUSD S    1000 96.92
0D14:56:01.127310000 SGDUSD S    2000 98.83
0D14:56:01.129311000 SGDUSD B    1000 94.94
0D14:56:01.130311000 AUDUSD B    5000 94.52
...
```

Rename the new tplog to the convention expected by the TP and RDB (in this example we rename to `sym2014.05.03`) . Restart both processes so that they write and read respectively to the correct tplog. Upon restarting, the RDB should read this tplog in and replay the tables correctly.

4.2 Illegal operations on keyed tables

Consider a Real-Time Engine designed to keep track of trading account position limits. This could take the form of a keyed table, where sym is an account name:

```
q) `sym xkey `accounts
`accounts
q)accounts
sym         | time                          curr   action limit
------------| ----------------------------------------------
fgAccount   | 2014.05.04D10:27:00.288697000 AUDJPY insert 5000000
pbAcc       | 2014.05.04D10:27:00.291699000 GBPUSD insert 1000000
ACCOUNT0023 | 2014.05.04D10:27:01.558332000 SGDUSD insert 1000000
```

If we wanted this keyed table to be recoverable within this process, we would publish any changes to the table via the tickerplant and have a customised `upd` function defined locally in the RTE:

```
upd:{[t;x]
  $[t~`accounts;
      $[`insert~a:first x`action;[t insert x];
        `update~a;@[`.;t;,;x];
        `delete~a;@[`.;t;:;delete from value[t] where sym=first x`sym ];
        '`unknownaction];
      t insert x];
      };
```

Here we have three operations we can perform on the accounts table: insert, update and delete. We wish to record the running of these operations in the tplog, capture in the RDB (in an unkeyed version of the table), and perform customised actions in the RTE. We create a publish function to publish the data to the TP. The TP will publish to both the RDB and RTE, and the `upd` function will then be called locally on each. Assuming TP is running on same machine on port 5010, with no user access credentials required, we can define a function named `pub` on the RTE which will publish data from the RTE to the TP where it can be logged and subsequently re-published to the RDB and RTE:

```
.tp.h:hopen`:localhost:5010;
pub:{[t;x]
  neg[.tp.h](`upd;t;x);
  h""
 };
```

Example usage on the RTE:

```
// - insert new account
q)pub[`accounts; enlist
sym`time`curr`action`limit!(`ACCOUNT0024;.z.p;`SGDUSD;`insert;1000000)];
q)accounts
sym        | time                          curr   action limit
-----------| ----------------------------------------------
fgAccount  | 2014.05.04D10:51:49.288168000 AUDJPY insert 5000000
pbAcc      | 2014.05.04D10:51:49.291168000 GBPUSD insert 1000000
ACCOUNT0023| 2014.05.04D10:51:50.950002000 SGDUSD insert 1000000
ACCOUNT0024| 2014.05.04D10:54:41.796915000 SGDUSD insert 1000000

// - update the limit on account ACCOUNT0024
q)pub[`accounts; enlist
`sym`time`curr`action`limit!(`ACCOUNT0024;.z.p;`SGDUSD;`update;7000000)]
q)accounts
sym        | time                          curr   action limit
-----------| ----------------------------------------------
fgAccount  | 2014.05.04D10:51:49.288168000 AUDJPY insert 5000000
pbAcc      | 2014.05.04D10:51:49.291168000 GBPUSD insert 1000000
ACCOUNT0023| 2014.05.04D10:51:50.950002000 SGDUSD insert 1000000
ACCOUNT0024| 2014.05.04D11:05:30.557228000 SGDUSD update 7000000

// - delete account ACCOUNT0024 from table
q)pub[`accounts; enlist
`sym`time`curr`action`limit!(`ACCOUNT0024;.z.p;`SGDUSD;`delete;1000000)]
q)accounts
sym         | time                           curr   action limit
------------| -----------------------------------------------
fgAccount   | 2014.05.04D10:27:00.288697000 AUDJPY insert 5000000
pbAcc       | 2014.05.04D10:27:00.291699000 GBPUSD insert 1000000
ACCOUNT0023 | 2014.05.04D10:27:01.558332000 SGDUSD insert 1000000
```

Each action will be recorded in an unkeyed table in the RDB, resulting in the following:

```
q)accounts
sym         time                          curr   action limit
-------------------------------------------------------------
fgAccount   2014.05.04D10:27:00.288697000 AUDJPY insert 5000000
pbAcc       2014.05.04D10:27:00.291699000 GBPUSD insert 1000000
ACCOUNT0023 2014.05.04D10:27:01.558332000 SGDUSD insert 1000000
ACCOUNT0024 2014.05.04D10:54:41.796915000 SGDUSD insert 1000000
ACCOUNT0024 2014.05.04D11:05:30.557228000 SGDUSD update 7000000
ACCOUNT0024 2014.05.04D11:05:30.557228000 SGDUSD delete 1000000
```

The order in which logfile messages are replayed is hugely important in this case. All operations on this table should be made via the tickerplant so that everything is logged and order is maintained. After the three operations above, the tplog will have the following lines appended:

```
q)get`:TP_2014.05.04
(`upd;`accounts; +`sym`time`curr`action`limit!(`ACCOUNT0024;
2014.05.04D10:54:41.796915000;`SGDUSD;`insert;1000000);
(`upd;`accounts; +`sym`time`curr`action`limit!(`ACCOUNT0024;
2014.05.04D11:05:30.557228000;`SGDUSD;`update;7000000);
(`upd;`accounts; +`sym`time`curr`action`limit!(`ACCOUNT0024;
2014.05.04D11:05:30.557228000;`SGDUSD;`delete;1000000);
```

Replaying this logfile will recover this table. If we wanted every operation to this table to be recoverable from a logfile, we would need to publish each operation. However, manual user actions that are not recorded in the tplog can cause errors when replaying. Consider the following table on the RTE:

```
q)accounts
sym        | time                          curr   action limit
-----------| ----------------------------------------------

// - insert new account
q)pub[`accounts; enlist
`sym`time`curr`action`limit!(`ACCOUNT0024;.z.p;`SGDUSD;`insert;1000000)]
;
sym        | time                          curr   action limit
-----------| ----------------------------------------------
ACCOUNT0024| 2014.05.04D10:54:41.796915000 SGDUSD insert 1000000

// - delete this entry from the in-memory table without publishing to
the TP
q)delete from `accounts where sym=` ACCOUNT0024
q)accounts
sym        | time                          curr   action limit
-----------| ----------------------------------------------

// - insert new account again
q)pub[`accounts; enlist
`sym`time`curr`action`limit!(`ACCOUNT0024;.z.p;`SGDUSD;`insert;1000000)]
;
sym        | time                          curr   action limit
-----------| ----------------------------------------------
ACCOUNT0024| 2014.05.04D10:54:41.796915000 SGDUSD insert 1000000
```

Only two of these three actions were sent to the TP so only these two messages were recorded in the tplog:

```
q)get`:TP_2014.05.04
(`upd;`accounts; +`sym`time`curr`action`limit!(`ACCOUNT0024;
2014.05.04D10:54:41.796915000;`SGDUSD;`insert;1000000);
(`upd;`accounts; +`sym`time`curr`action`limit!(`ACCOUNT0024;
2014.05.04D10:54:41.796915000;`SGDUSD;`insert;1000000);
```

Replaying the tplog, the q process tries to perform an insert to a keyed table, but there is a restriction that the same key cannot be inserted in the table twice. The delete that was performed manually between these two steps was omitted:

```
q)-11!`:TP_2014.05.04
'insert
```

4.3 Recovering from q errors during replay

As seen in the previous section, replaying a tplog can result in an error even if the log file is uncorrupted. We can utilise error trapping to isolate any rows in the tplog which cause an error while transferring the other, error-free rows into a new log file. The problematic tplog lines will be stored in a variable where they can be analysed to determine the next course of action.

```
old:`:2014.05.03;
new:`TP_2014.05.03_new;
//write an empty list to the new logfile
new set () ;
//open handle to new log file so we can write to it
 h:hopen new;
//save original upd fn
updOld:upd;
//create a variable to hold messages that are throwing errors
baddata:();
//redefine upd with error trapping
upd:{[t;x] .[{[t;x]
   //Try to run the original upd fn on the log line. If successful,
write to new logfile
            updOld[t;x];
                  h enlist(`upd;t;x) } ;
                  //input params
                  (t;x);
                  //Error catch. Log error and append to variable
                  {[t;x;errmsg](0N!"error on upd: ",errmsg
                                    baddata,:enlist(t;x))
                        }[t;x;]
                   ]
      };
```

We now replay the original tplog:

```
q) -11!old
2
q)accounts
sym        | time                          curr   action limit
-----------| ----------------------------------------------
ACCOUNT0024| 2014.05.04D10:54:41.796915000 SGDUSD insert 1000000
```

Here, both lines of the tplog have been replayed, but only one has actually been executed. The second insert has been appended to the `baddata` variable:

```
q) baddata
(`upd;`accounts; +`sym`time`curr`action`limit!(`ACCOUNT0024;
2014.05.04D10:54:41.796915000;`SGDUSD;`insert;1000000);
```

We may wish to write the bad data to its own logfile:

```
(`:tp_2014.05.03_new) set enlist each `upd,/: baddata;
```

5 CONCLUSION

This paper discussed tickerplant log files and their importance within kdb+tick systems. They act as a record of every message that passes through the system. In the event of a real-time subscriber process crashing, it can connect to a tickerplant and replay the log file using the `-11!` keyword, restoring all intra-day data. Section 3 examined the various parameters that can be supplied to `-11!`.

The log file can, however, become corrupted. For instance, the tickerplant process might die in the process of writing to the file, or the disk where the file is being written could fill up. In this scenario, a simple replay of the log file is not possible. However, using various combinations of `-11!`, we can at least recover any data up until the point of corruption. This is described in section 4.1.

Sections 4.2 and 4.3 discussed scenarios in which replaying a log file could result in an error, even if the file is uncorrupted. In this case, we utilised error trapping in order to isolate any rows of data causing errors whilst successfully replaying any valid rows. The problematic data could then be viewed in isolation in order to determine the best course of action.

All tests were run using kdb+ version 3.1 (2014.03.27).