AquaQ TorQ

email:
support@aquaq.co.uk

web:
www.aquaq.co.uk

Revision History

Revision	Date	Author(s)	Description
1.0	2014.02.12	AquaQ	First version released
1.1	2014.03.11	Tobias Harper	Housekeeping process information added
1.1	2014.03.25	Glen Smith	html.q and Monitor HTML5 front end information added
1.1	2014.04.02	Louise Belshaw	compress.q utitlity information added
1.1	2014.04.02	Andrew Steele	Filealerter process information added
1.2	2014.09.19	Deividas Macijauskas	Added info on connections to external (non-TorQ) processes
2.0	2015.01.10	Glen Smith	Reporter process information added
2.0	2015.01.10	Ian Kilpatrick	Email functionality added
2.0	2015.01.10	Andrew Steele	RDB, WDB, dbwriteutils and subscription logic added
2.0	2015.01.10	Mark Rooney	Monitoring checks and dataloader info added

Contents

Chapter 1

Company Overview

AquaQ Analytics Limited is a provider of specialist data management, data analytics and data mining services. We also provide strategic advice, training and consulting services in the area of market-data collection to clients within the capital markets sector. Our domain knowledge, combined with advanced analytical techniques and expertise in best-of-breed technologies, helps our clients get most out of their data.

The company is currently focussed on four key areas, all of which are conducted either on client site or near-shore:

- Kdb+ Consulting Services: Development, Training and Support. We also provide online kdb+ training[1] and kdb+ remote support[2].
- Real Time GUI Development Services
- SAS Analytics Services
- Providing IT consultants to investment banks with Java, .NET and Oracle experience

The company currently has a headcount of 40 consisting of both full time employees and contractors and is actively hiring additional resources. Some of these resources are based full-time on client site while others are involved in remote/near-shore development and support work from our Belfast headquarters.

[1] http://training.aquaq.co.uk
[2] http://support.aquaq.co.uk

Chapter 2

Overview

2.1 What is kdb+?

kdb+ is the market leading timeseries database from Kx Systems[1]. kdb+ is used predominently in the Financial Services sector to capture, process and analyse billions of records on a daily basis, with Kx counting almost all of the top tier investment banks as customers. kdb+ incorporates a programming language, q, which is known for its performance and expressive power. Given the unsurpassed data management and analytical capabilities of kdb+, the applicability of kdb+ technology extends beyond the financial domain into any sector where rapid pre-built or adhoc analysis of large datasets is required. Other sectors which have good use cases for kdb+ include utilities, pharmaceuticals, telecoms, manufacturing, retail and any sector utilising telemetry or sensor data.

2.2 What is AquaQ TorQ?

AquaQ TorQ is a framework which forms the basis of a production kdb+ system by implementing some core functionality and utilities on top of kdb+, allowing developers to concentrate on the application business logic. We have incorporated as many best practices as possible, with particular focus on performance, process management, diagnostic information, maintainability and extensibility. We have kept the code as readable as possible using descriptive comments, error messages, function names and variable names. Wherever possible, we have tried to avoid re-inventing the wheel and instead have used contributed code from code.kx.com (either directly or modified). All code sections taken from code.kx.com are referenced in this document.

[1] `www.kx.com`

AquaQ TorQ can be extended or modified as required. We have chosen some default behaviour, but it can all be overridden. The features of AquaQ TorQ are:

- Process Management: Each process is given a type and name. By default these are used to determine the code base it loads, the configuration loaded, log file naming and how it reports itself to discovery services. Whenever possible we have tried to ensure that all default behaviour can be overridden at the process type level, and further at the process name level.
- Code Management: Processes can optionally load common or process type/name specific code bases. All code loading is error trapped.
- Configuration Management: Configuration scripts can be loaded as standard and with specific process type/name configuration overriding default values. Configuration scripts are loaded in a specific order; default, then process type specific, then process name specific. Values loaded last will override values loaded previously.
- Usage Logging: All process usage is logged to a single text log file and periodically rolled. Logging includes opening/closing of connections, synchronous and asynchronous queries and functions executed on the timer. Logged values include the request, the details of where it came from, the time it was received, the time it took, memory usage before and after, the size of the result set, the status and any error message.
- Incoming and Outgoing Connection Management: Incoming (client) and outgoing (server) connections are stored and their usage monitored through query counts and total data size counts. Connections are stored and retrieved and can be set to automatically be re-opened as required. The password used for outgoing connections can be overridden at default, process type and process name level.
- Access Controls: Basic access controls are provided, and could be extended. These apply restrictions on the IP addresses of remote connections, the users who can access the process, and the functions that each user can execute. A similar hierarchical approach is used for access control management as for configuration management.
- Timer Extensions: Mechanism to allow multiple functions to be added to the timer either on a repeating or one-off basis. Multiple re-scheduling algorithms supplied for repeating timers.
- Standard Out/Error Logging: Functions to print formatted messages to standard out and error. Hooks are provided to extend these as required, e.g. publication to centralised logging database. Standard out and error are redirected to appropriately named, timestamped and aliased log files, which are periodically rolled.
- Error Handling: Different failure options are supplied in case code fails to load; either exit upon failure, stop at the point of failure or trap and continue.

- Visualisation: Utilities to ease GUI development using websockets and HTML5.
- Documentation and Development Tools: Functionality to document the system is built into AquaQ TorQ, and can be accessed directly from every q session. Developers can extend the documentation as they add new functions. Functionality for searching for functions and variables by name and definition is provided, and for ordering variables by memory usage. The standard help.q from code.kx is also included.
- Utilities: We intend to build out and add utilities as we find them to be suitably useful and generic. So far we have:
    1. Caching: allows a result set cache to be declared and result sets to be stored and retrieved from the cache. Suitable for functions which may be run multiple times with the same parameters, with the underlying data not changing in a short time frame;
    2. Timezone Handling: derived from code.kx, allows conversion between timestamps in different timezones;
    3. Email: an library to send emails;
    4. Async Messaging: allows easy use of advanced async messaging methods such as deferred synchronous communication and async communication using postback functions;
    5. Heartbeating: each process can be set to publish heartbeats, and subscribe to and manage heartbeats from other processes in the environment;
    6. Data Loading: utility wrapper around .Q.fsn to read a data file from disk, manipulate it and write it out in chunks;
    7. Subscriptions: allow processes to dynamically detect and subscribe to datasources;
    8. Tickerplant Log File Recovery: recover as many messages as possible from corrupt log files;
    9. Database Writing: utility functions for writing to, sorting and parting on disk databases;
    10. Compression: allows compression of a database. This can be performed using a set of parameters for the entire database, but also gives the flexibilty of compressing user-specified tables and/or columns of those tables with different parameters if required, and also offers decompression.

AquaQ TorQ will wrap easily around kdb+tick and therefore around any tickerplant, RDB, HDB or real time processing application. We currently have several customised processes of our own:

- Discovery Service: Every process has a type, name and set of available attributes, which are used by other processes to connect to it. The Discovery Service is a central point that can be used to find other available processes. Client processes can subscribe to updates from the discovery service as new processes become available- the discovery service will notify its subscribers, which can then use the supplied hook to implement required behavior e.g. connect to the newly available process;
- Gateway: A fully synchronous and asynchronous gateway is provided. The gateway will connect to a defined list of process types (can be homogenous or heterogeneous processes) and will route queries across them according to the priority of received requests. The routing algorithms can be easily modified e.g. give priority to user X, or only route queries to processes which exist in the same data centre or geographical region to avoid the WAN (this would entail using the process attributes). The gateway can either return the result to the client back down the same handle, or it can wrap it in a callback function to be invoked on the client;
- Real Time Database (RDB): A customized version of the kdb+tick RDB, to allow dynamic tickerplant subscriptions, reloading of multiple HDBs using authenticated connections, and customized end-of-day save downs. The RDB, WDB and tickerplant log replay share a common code base to ensure that a save-down modification to a table is applied across each of these processes.
- Write Database (WDB): The job of a WDB is to write data to disk rather than to serve client queries. WDBs usually write data out periodically throughout the day, and are useful when there is too much data to fit into memory and/or the end of day save operation needs to be speeded up. The concept is based on w.q[2]
- Tickerplant Log Replay: A process for replaying tickerplant log files to create on-disk data sets. Extended features are provided for only replaying subsets of log files (by message number and/or table name), replaying in chunks, invoking bespoke final behaviour etc.;
- Reporter: The Reporter Process runs defined reports (q queries or parameterized functions) against specific database or gateways on a schedule. The results are retrieved and processed. Processing can be user defined, or can be a standard operation such as writing the data to disk, or emailing the results to a list of recipients. This can be useful for running system checks or generating management reports.
- Housekeeping: A process to undertake housekeeping tasks periodically, such as compressing and removing files that are no longer used. Housekeeping looks up a file of instructions and performs maintenance tasks on directories accordingly. Features allow selective file deletion and zipping according to file age, including

[2] `http://code.kx.com/wiki/Cookbook/w.q`

a search string parameter and the ability to exclude items from the search. The process can be scheduled, or run immediately from the command line and can be extended as required to incorporate more tasks.

- File Alerter: A process to periodically scan a set of directories and execute a function based on the availability of a file. This is useful where files may arrive to the system during the day and must be acted upon (e.g. files are uploaded to a shared directory by users/clients). The functions to execute are defined by the user and the whole process is driven by a csv file detailing the file to search for, the function to execute and, optionally, a directory to move the file to after it has been processed.
- Monitor: A basic monitoring process which uses the Discovery Service to locate the other processes within the system, listens for heartbeats, and subscribes for log messages. This should be extended as required but provides a basic central point for system health checks;
- Kill: A process used to kill other processes, optionally using the Discovery Service to locate them.

2.3 A Large Scale Data Processing Platform

One of the key drivers behind TorQ development has been to ensure all the tools necessary to build a large scale data processing platform are available. kdb+tick[3] provides the basic building blocks, and a standard set-up usually looks something like this:

[3] `http://code.kx.com/wsvn/code/kx/kdb+tick`

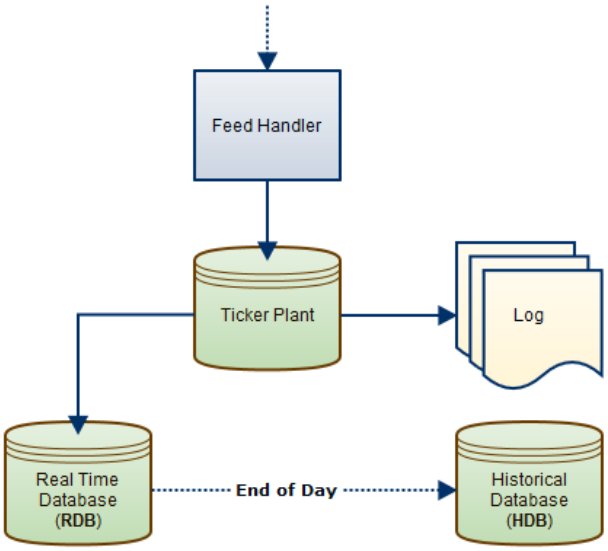

Figure 2.1: Simple kdb+tick Setup

However, in reality it is usually more complicated. A larger scale architecture serving large numbers of client queries and receiving data from multiple sources may look like this:

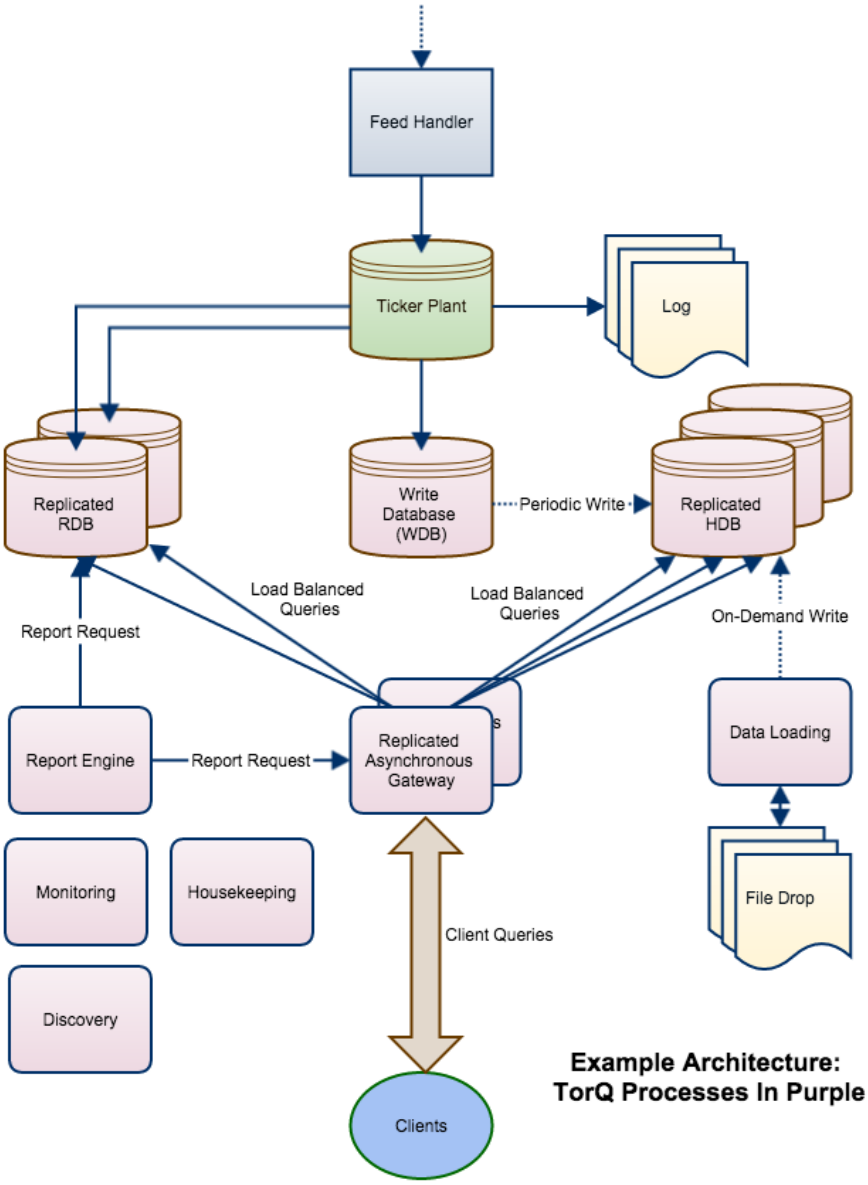

Figure 2.2: Production Data Capture

A common practice is to use a gateway (section 8.2) to manage client queries across back-end processes. The gateway can load balance across processes and make failures transparent to the client. If the clients access the gateway with asynchronous calls, then the gateway can serve many requests at once and additionally implement client queuing algorithms.

Other common production features include:

- A modified version of the RDB (section 8.3) which does different operations at end-of-day, reloads multiple HDB processes etc.
- A Write Database (section 8.4) which receives data from the tickerplant and periodically writes it to disk. WDBs are used when there is too much data in a day to fit into memory and/or to speed up the end-of-day rollover job
- Processes that load data from other sources either into the HDB directly or to the RDB potentially via the tickerplant (section 6.8). The data may be dropped in specific locations which have to be monitored (section 8.7)
- A Reporting Engine (section 8.8) to run periodic reports and do something with the result (e.g. generate an xls file from the database and email it to senior management). Reporting engines can also be used to run periodic checks of the system
- A Discovery Service (section 8.1) to allow processes to locate each other, and to allow processes to dynamically register availability and push notifications around the system.
- Basic Monitoring (section 8.9) of process availability
- Housekeeping (section 8.6) to ensure log files are tidied up, tickerplant log files are compressed/moved in a timely fashion etc.

2.4 Do I Really Have to Read This Whole Document?

Hopefully not. The core of AquaQ TorQ is a script called torq.q and we have tried to make it as descriptive as possible, so perhaps that will suffice. The first place to look will be in the config files, the main one being $KDBCONFIG/settings/default.q. This should contain a lot of information on what can be modified. In addition:

- We have added a load of usage information:

```
aquaq$ q torq.q -usage
KDB+ 3.1 2013.10.08 Copyright (C) 1993-2013 Kx Systems

General:
 This script should form the basis of a production kdb+ environment.
 It can be sourced from other files if required, or used as a launch script
     before loading other files/directories using either -load or -loaddir
     flags
... etc ...
```

  If sourcing from another script there are hooks to modify and extend the usage information as required.

- We have some pretty extensive logging:

```
aquaq$ q torq.q -p 9999 -debug
KDB+ 3.1 2013.10.08 Copyright (C) 1993-2013 Kx Systems
```



```
2013.11.05D12:22:42.597500000|aquaq|torq.q_3139_9999|INF|init|trap mode (
    initialisation errors will be caught and thrown, rather than causing an
    exit) is set to 0
2013.11.05D12:22:42.597545000|aquaq|torq.q_3139_9999|INF|init|stop mode (
    initialisation errors cause the process loading to stop) is set to 0
2013.11.05D12:22:42.597810000|aquaq|torq.q_3139_9999|INF|init|attempting to
    read required process parameters proctype,procname from file /torqhome/
    config/process.csv
2013.11.05D12:22:42.598081000|aquaq|torq.q_3139_9999|INF|init|read in process
    parameters of proctype=hdb; procname=hdb1
2013.11.05D12:22:42.598950000|aquaq|hdb1|INF|fileload|config file /torqhome/
    config/default.q found
... etc ...
```

- We have added functionality to find functions or variables defined in the session, and also to search function definitions.

```
q).api.f`max
name                | vartype   namespace public descrip              ..
--------------------| ------------------------------------------------..
maxs                | function  .q        1      ""                   ..
mmax                | function  .q        1      ""                   ..
.clients.MAXIDLE    | variable  .clients  0      ""                   ..
.access.MAXSIZE     | variable  .access   0      ""                   ..
.cache.maxsize      | variable  .cache    1      "The maximum size in ..
.cache.maxindividual| variable  .cache    1      "The maximum size in ..
max                 | primitive           1      ""                   ..

q)first 0!.api.p`.api
name     | `.api.f
vartype  | `function
namespace| `.api
public   | 1b
descrip  | "Find a function/variable/table/view in the current process"
params   | "[string:search string]"
return   | "table of matching elements"

q).api.p`.api
name        | vartype  namespace public descrip                       ..
------------| --------------------------------------------------------..
.api.f      | function .api      1      "Find a function/variable/tabl..
.api.p      | function .api      1      "Find a public function/variab..
.api.u      | function .api      1      "Find a non-standard q public ..
.api.s      | function .api      1      "Search all function definitio..
.api.find   | function .api      1      "Generic method for finding fu..
.api.search | function .api      1      "Generic method for searching ..
.api.add    | function .api      1      "Add a function to the api des..
.api.fullapi| function .api      1      "Return the full function api ..
```

- We have incorporated help.q.

```
q)help`
adverb    | adverbs/operators
attributes| data attributes
cmdline   | command line parameters
data      | data types
define    | assign, define, control and debug
dotz      | .z locale contents
errors    | error messages
save      | save/load tables
```



```
syscmd    | system commands
temporal  | temporal - date & time casts
verbs     | verbs/functions
```

- We have separated and commented all of our config:

```
aquaq$ head config/default.q
/- Default configuration - loaded by all processes

/- Process initialisation
\d .proc
loadcommoncode:1b       /- whether to load the common code defined at
                        /- ${KDBCODE}/common
loadprocesscode:0b      /- whether to load the process specific code defined at
                        /- ${KDBCODE}/{process type}
loadnamecode:0b         /- whether to load the name specific code defined at
                        /- ${KDBCODE}/{name of process}
loadhandlers:1b         /- whether to load the message handler code defined at
                        /- ${KDBCODE}/handlers
logroll:1b              /- whether to roll the std out/err logs daily
... etc ...
```

2.5 Operating System and kdb+ Version

AquaQ TorQ has been built and tested on the linux and OSX operating systems though as far as we are aware there is nothing that would make this incompatible with Solaris or Windows. It has also been tested with kdb+ 3.1 and 2.8. Please report any incompatibilities with other kdb+ versions or operating systems.

2.6 License

This code is released under the MIT license[4].

[4] http://opensource.org/licenses/MIT

Chapter 3

Getting Started

kdb+ is very customisable. Customisations are contained in q scripts (.q files), which define functions and variables which modify the behaviour of a process. Every q process can load a single q script, or a directory containing q scripts and/or q data files. Hooks are provided to enable the programmer to apply a custom function to each entry point of the process (.z.p*), to be invoked on the timer (.z.ts) or when a variable in the top level namespace is amended (.z.vs). By default none of these hooks are implemented.

We provide a codebase and a single main script, torq.q. torq.q is essentially a wrapper for bespoke functionality which can load other scripts/directories, or can be sourced from other scripts. Whenever possible, torq.q should be invoked directly and used to load other scripts as required. torq.q will:

- ensure the environment is set up correctly;
- define some common utility functions (such as logging);
- execute process management tasks, such as discovering the name and type of the process, and re-directing output to log files;
- load configuration;
- load the shared code based;
- set up the message handlers;
- load any required bespoke scripts.

The behavior of torq.q is modified by both command line parameters and configuration. We have tried to keep as much as possible in configuration files, but if the parameter either has a global effect on the process or if it is required to be known before the configuration is read, then it is a command line parameter.

3.1 Using torq.q

torq.q can be invoked directly from the command line and be set to source a specified file or directory. torq.q requires the 3 environment variables to be set (see section 3.2). If using a unix environment, this can be done with the setenv.sh script. To start a process in the foreground without having to modify any other files (e.g. process.csv) you need to specify the type and name of the process as parameters. An example is below.

```
$ . setenv.sh
$ q torq.q -debug -proctype testproc -procname test1
```

To load a file, do:

```
$ q torq.q -load myfile.q -debug -proctype testproc -procname test1
```

It can also be sourced from another script. If this is the case, some of the variables can be overridden, and the usage information can be modified or extended. Any variable that has a definition like below can be overridden from the loading script.

```
myvar:@[value;`myvar;1 2 3]
```

The available command line parameters are:

Cmd Line Param	Description
-procname x -proctype y	The process name and process type
-procfile x	The name of the file to get the process information from
-load x [y..z]	The files or database directory to load
-loaddir x [y..z]	Load all .q, .k files in specified directories
-trap	Any errors encountered during initialization when loading external files will be caught and logged, processing will continue
-stop	Stop loading the file if an error is encountered but do not exit
-noredirect	Do not redirect std out/std err to a file (useful for debugging)
-noredirectalias	Do not create an alias for the log files (aliases drop any suffix e.g. timestamp suffix)
-noconfig	Do not load configuration
-nopi	Reset the definition of .z.pi to the initial value (useful for debugging)
-debug	Equivalent to [-nopi -noredirect]
-usage	Print usage info and exit

Table 3.1: torq.q Command Line Parameters

In addition any process variable in a namespace (.*.*) can be overridden from the command line. Any value supplied on the command line will take priority over any other predefined value (.e.g. in a configuration or wrapper). Variable names should be supplied with full qualification e.g. -.servers.HOPENTIMEOUT 5000.

3.2 Environment Variables

Three environment variables are required:

Environment Variable	Description
KDBCONFIG	The base configuration directory
KDBCODE	The base code directory
KDBLOGS	Where standard out/error and usage logs are written

Table 3.2: Required Environment Variables

torq.q will check for these and exit if they are not set. If torq.q is being sourced from another script, the required environment variables can be extended by setting .proc.envvars before loading torq.q.

3.3 Process Identification

At the crux of AquaQ TorQ is how processes identify themselves. This is defined by two variables - .proc.proctype and .proc.procname which are the type and name of the process respectively. These two values determine the code base and configuration loaded, and how they are connected to by other processes.

The most important of these is the proctype. It is up to the user to define at what level to specify a process type. For example, in a production environment it would be valid to specify processes of type "hdb" (historic database) and "rdb" (real time database). It would also be valid to segregate a little more granularly based on approximate functionality, for example "hdbEMEA" and "hdbAmericas". The actual functionality of a process can be defined more specifically, but this will be discussed later. The procname value is used solely for identification purposes. A process can determine its type and name in one of four ways:

1. From the process file in the default location of $KDBCONFIG/process.csv;
2. From the process file defined using the command line parameter -procfile;
3. Using the command line parameters -proctype and -procname;
4. By defining .proc.proctype and .proc.procname in a script which loads torq.q.

For options 3 and 4, both parameters must be defined using that method or neither will be used (the values will be read from the process file). The process file has format as below.

```
aquaq$ cat config/process.csv
host,port,proctype,procname
aquaq,9997,rdb,rdb_europe_1
aquaq,9998,hdb,hdb_europe_1
aquaq,9999,hdb,hdb_europe_2
```

The process will read the file and try to identify itself based on the host and port it is started on. The host can either be the value returned by .z.h, or the ip address of the server. If the process can not automatically identify itself it will exit.

3.4 Logging

By default, each process will redirect output to a standard out log and a standard error log, and create aliases for them. These will be rolled at midnight on a daily basis. They are all written to the $KDBLOGS directory. The log files created are:

Log File	Description
out_[procname]_[date].log	Timestamped out log
err_[procname]_[date].log	Timestamped error log
out_[procname].log	Alias to current log log
err_[procname].log	Alias to current error log

Table 3.3: Log Files

The date suffix can be overridden by modifying the .proc.logtimestamp function and sourcing torq.q from another script. This could, for example, change the suffixing to a full timestamp.

3.5 Configuration Loading

The process configuration is contained in q scripts, and stored in the $KDBCONFIG directory. Each process tries to load all the configuration it can find. Each process will attempt to load three configuration files in the below order-

- default.q: default configuration loaded by all processes. In a standard installation this should contain the superset of customisable configuration, including comments;
- [proctype].q: configuration for a specific process type;

- [procname].q: configuration for a specific named process.

The only one which should always be present is default.q. Each of the other scripts can contain a subset of the configuration variables, which will override anything loaded previously. Configuration is loaded before code.

3.6 Code Loading

Code is loaded from the $KDBCODE directory. There is also a common codebase, a codebase for each process type, and a code base for each process name, contained in the following directories and loaded in this order:

- $KDBCODE/common: shared codebase loaded by all processes;
- $KDBCODE/[proctype]: code for a specific process type;
- $KDBCODE/[procname]: code for a specific process name;

For any directory loaded, the load order can be specified by adding order.txt to the directory. order.txt dictates the order that files in the directory are loaded. If a file is not in order.txt, it will still be loaded but after all the files listed in order.txt have been loaded.

Additional directories can be loaded using the -loaddir command line parameter.

3.7 Initialization Errors

Initialization errors can be handled in different ways. The default action is any initialization error causes the process to exit. This is to enable fail-fast type conditions, where it is better for a process to fail entirely and immediately than to start up in an indeterminate state. This can be overridden with the -trap or -stop command line parameters. With -trap, the process will catch the error, log it, and continue. This is useful if, for example, the error is encountered loading a file of stored procedures which may not be invoked and can be reloaded later. With -stop the process will halt at the point of the error but will not exit. Both -stop and -trap are useful for debugging.

Chapter 4

Message Handlers

There is a separate code directory containing message handler customizations. This is found at $KDBCODE/handlers. Much of the code is derived from Simon Garland's contributions to code.kx[1].

Every external interaction with a process goes through a message handler, and these can be modified to, for example, log or restrict access. Passing through a bespoke function defined in a message handler will add extra processing time and therefore latency to the message. All the customizations we have provided aim to minimise additional latency, but if a bespoke process is latency sensitive then some or all of the customizations could be switched off. We would argue though that generally it is better to switch on all the message handler functions which provide diagnostic information, as for most non-latency sensitive processes (HDBs, Gateways, some RDBs etc.) the extra information upon failure is worth the cost. The message handlers can be globally switched off by setting .proc.loadhandlers to 0b in the configuration file.

[1] http://code.kx.com/wiki/Contrib/UsingDotz

Script	NS	Diag	Function	Modifies
logusage.q	.usage	Y	Log all client interaction to an ascii log file and/or in-memory table. Messages can be logged before and after they are processed. Timer calls are also logged. Exclusion function list can be applied to .z.ps to disable logging of asynchronous real time updates	pw, po, pg, ps, pc, ws, ph, pp, pi, exit, timer
controlaccess.q	.access	N	Restrict access for set of users/user groups to a list of functions, and from a defined set of servers	pw, pg, ps, ws, ph, pp, pi
trackclients.q	.clients	Y	Track client process details including then number of requests and cumulative data size returned	po, pg, ps, ws, pc
trackservers.q	.servers	Y	Discover and track server processes including name, type and attribute information. This also contains the core of the code which can be used in conjunction with the discovery service.	pc, timer
zpsignore.q	.zpsignore	N	Override async message handler based on certain message patterns	ps

Table 4.1: Message Handler Scripts

Each customization can be turned on or off individually from the configuration file(s). Each script can be extensively customised using the configuration file. Example customization for logusage.q, taken from $KDBCONFIG/settings/default.q is below. Please see default.q for the remaining configuration of the other message handler files.

```
/- Configuration used by the usage functions - logging of client interaction
\d .usage
enabled:1b              /- whether the usage logging is enabled
logtodisk:1b            /- whether to log to disk or not
logtomemory:1b          /- write query logs to memory
ignore:1b               /- check the ignore list for functions to ignore
ignorelist:(`upd;"upd") /- the list of functions to ignore in async calls
flushtime:1D00          /- default value for how long to persist the
                        /- in-memory logs. Set to 0D for no flushing
suppressalias:0b        /- whether to suppress the log file alias creation
logtimestamp:{[].z.d}   /- function to generate the log file timestamp suffix
LEVEL:3                 /- log level. 0=none;1=errors;2=errors+complete
                        /- queries;3=errors+before a query+after
logroll:1b              /- Whether or not to roll the log file
                        /- automatically (on a daily schedule)
```


4.1 logusage.q

logusage.q is probably the most important of the scripts from a diagnostic perspective. It is a modified version of the logusage.q script on code.kx.

In its most verbose mode it will log information to an in-memory table (.usage.usage) and an on-disk ASCII file, both before and after every client interaction and function executed on the timer. These choices were made because:

- logging to memory enables easy interrogation of client interaction;
- logging to disk allows persistence if the process fails or locks up. ASCII text files allow interrogation using OS tools such as vi, grep or tail;
- logging before a query ensures any query that adversely effects the process is definitely captured, as well as capturing some state information before the query execution;
- logging after a query captures the time taken, result set size and resulting state;
- logging timer calls ensures a full history of what the process is actually doing. Also, timer call performance degradation over time is a common source of problems in kdb+ systems.

The following fields are logged in .usage.usage:

Field	Description
time	Time the row was added to the table
id	ID of the query. Normally before and complete rows will be consecutive but it might not be the case if the incoming call invokes further external communication
timer	Execution time. Null for rows with status=b (before)
zcmd	.z handler the query arrived through
status	Query status. One of b, c or e (before, complete, error)
a	Address of sender. .dotz.ipa can be used to convert from the integer format to a hostname
u	Username of sender
w	Handle of sender
cmd	Command sent
mem	Memory statistics
sz	Size of result. Null for rows with status of b or e
error	Error message

Table 4.2: Usage Logging Fields

4.2 controlaccess.q

controlaccess.q is used to restrict client access to the process. It is modified version of controlaccess.q from code.kx. The script allows control of several aspects:

- the host/ip address of the servers which are allowed to access the process;
- definition of three user groups (default, poweruser and superuser) and the actions each group is allowed to do;
- the group(s) each user is a member of, and any additional actions an individual user is allowed/disallowed outside of the group permissions;
- the maximum size of the result set returned to a client.

The access restrictions are loaded from csv files. The permissions files are stored in $KDBCONFIG/permissions.

File	Description
*_hosts.csv	Contains hostname and ip address (patterns) for servers which are allowed or disallowed access. If a server is not found in the list, it is disallowed
*_users.csv	Contains individual users and the user groups they are are a member of
*_functions.csv	Contains individual functions and whether each user group is allowed to execute them. ; separated user list enables functions to be allowed by individual users

Table 4.3: Permissions Files

The permissions files are loaded using a similar hierarchical approach as for the configuration and code loading. Three files can be provided- default_*.csv, [proctype]_*.csv, and [procname]_*.csv. All of the files will be loaded, but permissions for the same entity (hostpattern, user, or function) defined in [procname]_*.csv will override those in [proctype]_*.csv which will in turn override [procname]_*.csv.

When a client makes a query which is refused by the permissioning layer, an error will be raised and logged in .usuage.usage if it is enabled.

4.3 trackclients.q

trackclients.q is used to track client interaction. It is a slightly modified version of trackclients.q from code.kx, and extends the functionality to handle interaction with the discovery service.

Whenever a client opens a connection to the q process, it will be registered in the .clients.clients table. Various details are logged, but from a diagnostic perspective the most important information are the client details, the number of queries it has run, the last time it ran a query, the number of failed queries and the cumulative size of results returned to it.

4.4 trackservers.q

trackservers.q is used to register and maintain handles to external servers. It is a heavily modified version of trackservers.q from code.kx. It is explained more in section 5.

4.5 zpsignore.q

zpsignore.q is used to check incoming async calls for certain patterns and to bypass all further message handler checks for messages matching the pattern. This is useful for handling update messages published to a process from a data source.

4.6 Diagnostic Reporting

The message handler modifications provide a wealth of diagnostic information including:

- the timings and memory usage for every query run on a process;
- failed queries;
- clients trying to do things they are not permissioned for;
- the clients which are querying often and/or regularly extracting large datasets;
- the number of clients currently connected;
- timer calls and how long they take.

Although not currently implemented, it would be straightforward to use this information to implement reports on the behaviour of each process and the overall health of the system. Similarly it would be straightforward to set up periodic publication to a central repository to have a single point for system diagnostic statistics.

Chapter 5

Connection Management

trackservers.q is used to register and maintain handles to external servers. It is a heavily modified version of trackservers.q from code.kx. All the options are described in the default config file. All connections are tracked in the .servers.SERVERS table. When the handle is used the count and last query time are updated.

```
q).servers.SERVERS
procname     proctype  hpup                              w  hits startp
                           lastp                           endp
                             attributes
-------------------------------------------------------------------------------
discovery1   discovery :aquaq:9996    0                                  2014.01.08
    D11:13:10.583056000                          ()!()
discovery2   discovery :aquaq:9995 6  0    2014.01.07D16:44:47.175757000 2014.01.07
    D16:44:47.174408000                          ()!()
rdb_europe_1 rdb       :aquaq:9998 12 0    2014.01.07D16:46:47.897910000 2014.01.07
    D16:46:47.892901000 2014.01.07D16:46:44.626293000 `datacentre`country!`essex`uk
rdb1         rdb       :aquaq:5011 7  0    2014.01.07D16:44:47.180684000 2014.01.07
    D16:44:47.176994000                          `datacentre`country!`essex`uk
rdb_europe_1 hdb       :aquaq:9997    0                                  2014.01.08
    D11:13:10.757801000                          ()!()
hdb1         hdb       :aquaq:9999    0                                  2014.01.08
    D11:13:10.757801000                          ()!()
hdb2         hdb       :aquaq:5013 8  0    2014.01.07D16:44:47.180684000 2014.01.07
    D16:44:47.176994000                          `datacentre`country!`essex`uk
hdb1         hdb       :aquaq:5012 9  0    2014.01.07D16:44:47.180684000 2014.01.07
    D16:44:47.176994000                          `datacentre`country!`essex`uk

q)last .servers.SERVERS
procname  | `hdb2
proctype  | `hdb
hpup      | `:aquaq:5013
w         | 8i
hits      | 0i
startp    | 2014.01.08D11:51:01.928045000
lastp     | 2014.01.08D11:51:01.925078000
endp      | 0Np
attributes| `datacentre`country!`essex`uk
```

5.1 Connections

Processes locate other processes based on their process type. The location is done either statically using the process.csv file or dynamically using a discovery service. It is recommended to use the discovery service as it allows the process to be notified as new processes become available.

The main configuration variable is .servers.CONNECTIONS, which dictates which process type(s) to create connections to. .servers.startup[] must be called to initialise the connections. When connections are closed, the connection table is automatically updated. The process can be set to periodically retry connections.

5.2 Process Attributes

Each process can report a set of attributes. When process A connects to process B, process A will try to retrieve the attributes of process B. The attributes are defined by the result of the .proc.getattributes function, which is by default an empty dictionary. Attributes are used to retrieve more detail about the capabilities of each process, rather than relying on the broad brush process type and process name categorization. Attributes can be used for intelligent query routing. Potential fields for attributes include:

- range of data contained in the process;
- available tables;
- instrument universe;
- physical location;
- any other fields of relevance.

5.3 Connection Passwords

The password used by a process to connect to external processes is retrieved using the .servers.loadpassword function call. By default, this will read the password from a txt file contained in $KDBCONFIG/passwords. A default password can be used, which is overridden by one for the process type, which is itself overridden by one for the process name. For greater security, the .servers.loadpassword function should be modified.

5.4 Retrieving and Using Handles

A function .servers.getservers is supplied to return a table of handle information. .servers.getservers takes five parameters:

- type-or-name: whether the lookup is to be done by type or name (can be either proctype or procname);
- types-or-names: the types or names to retrieve e.g. hdb;
- required-attributes: the dictionary of attributes to match on;
- open-dead-connections: whether to re-open dead connections;
- only-one: whether we only require one handle. So for example if 3 services of the supplied type are registered, and we have an open handle to 1 of them, the open handle will be returned and the others left closed irrespective of the open-dead-connections parameter.

.servers.getservers will compare the required parameters with the available parameters for each handle. The resulting table will have an extra column called attribmatch which can be used to determine how good a match the service is with the required attributes. attribmatch is a dictionary of (required attribute key) ! (Boolean full match; intersection of attributes).

```
q).servers.SERVERS
procname     proctype  hpup                           w hits startp
                           lastp                          endp attributes
-------------------------------------------------------------------------------
discovery1   discovery :aquaq:9996   0                                2014.01.08D11
    :51:01.922390000      ()!()
discovery2   discovery :aquaq:9995 6 0    2014.01.08D11:51:01.923812000 2014.01.08D11
    :51:01.922390000      ()!()
rdb_europe_1 rdb       :aquaq:9998   0                                2014.01.08D11
    :51:38.347598000      ()!()
rdb_europe_2 rdb       :aquaq:9997   0                                2014.01.08D11
    :51:38.347598000      ()!()
rdb1         rdb       :aquaq:5011 7 0    2014.01.08D11:51:01.928045000 2014.01.08D11
    :51:01.925078000      `datacentre`country!`essex`uk
hdb3         hdb       :aquaq:5012 9 0    2014.01.08D11:51:38.349472000 2014.01.08D11
    :51:38.347598000      `datacentre`country!`essex`uk
hdb2         hdb       :aquaq:5013 8 0    2014.01.08D11:51:01.928045000 2014.01.08D11
    :51:01.925078000      `datacentre`country!`essex`uk

/- pull back hdbs.  Leave the attributes empty
q).servers.getservers[`proctype;`hdb;()!();1b;f0b]
procname proctype lastp                         w hpup        attributes
                       attribmatch
-----------------------------------------------------------------------------
hdb3     hdb      2014.01.08D11:51:38.347598000 9 :aquaq:5012 `datacentre`country!`
    essex`uk ()!()
hdb2     hdb      2014.01.08D11:51:01.925078000 8 :aquaq:5013 `datacentre`country!`
    essex`uk ()!()

/- supply some attributes
q).servers.getservers[`proctype;`hdb;(enlist`country)!enlist`uk;1b;0b]
```



```
procname proctype lastp                          w hpup        attributes
                        attribmatch
--------------------------------------------------------------------------------
hdb3     hdb      2014.01.08D11:51:38.347598000 9 :aquaq:5012 `datacentre`country!`
    essex`uk (,`country)!,(1b;,`uk)
hdb2     hdb      2014.01.08D11:51:01.925078000 8 :aquaq:5013 `datacentre`country!`
    essex`uk (,`country)!,(1b;,`uk)
q).servers.getservers[`proctype;`hdb;`country`datacentre!`uk`slough;1b;0b]
procname proctype lastp                          w hpup        attributes
                        attribmatch
--------------------------------------------------------------------------------
hdb3     hdb      2014.01.08D11:51:38.347598000 9 :aquaq:5012 `datacentre`country!`
    essex`uk `country`datacentre!((1b;,`uk);(0b;`symbol$()))
hdb2     hdb      2014.01.08D11:51:01.925078000 8 :aquaq:5013 `datacentre`country!`
    essex`uk `country`datacentre!((1b;,`uk);(0b;`symbol$()))
```

.servers.getservers will try to automatically re-open connections if required.

```
q).servers.getservers[`proctype;`rdb;()!();1b;0b]
2014.01.08D12:01:06.023146000|aquaq|gateway1|INF|conn|attempting to open handle to :
    aquaq:9998
2014.01.08D12:01:06.023581000|aquaq|gateway1|INF|conn|connection to :aquaq:9998
    failed: hop: Connection refused
2014.01.08D12:01:06.023597000|aquaq|gateway1|INF|conn|attempting to open handle to :
    aquaq:9997
2014.01.08D12:01:06.023872000|aquaq|gateway1|INF|conn|connection to :aquaq:9997
    failed: hop: Connection refused
procname proctype lastp                          w hpup         attributes
                        attribmatch
--------------------------------------------------------------------------------
rdb1     rdb      2014.01.08D11:51:01.925078000 7 :aquaq:5011 `datacentre`country!`
    essex`uk ()!()

/- If we only require one connection, and we have one open,then it doesn't retry
    connections
q).servers.getservers[`proctype;`rdb;()!();1b;1b]
procname proctype lastp                          w hpup        attributes
                        attribmatch
--------------------------------------------------------------------------------
rdb1     rdb      2014.01.08D11:51:01.925078000 7 :aquaq:5011 `datacentre`country!`
    essex`uk ()!()
```

There are two other functions supplied for retrieving server details, both of which are based on .servers.getservers. .servers.gethandlebytype returns a single handle value, .servers.gethpupbytype returns a single host:port value. Both will re-open connections if there are not any valid connections. Both take two parameters:

- types: the type to retrieve e.g. hdb;
- selection-algorithm: can be one of any, last or roundrobin.

5.5 Connecting To Non-TorQ Processes

Connections to non-torq (external) processes can also be established. This is useful if you wish to integrate TorQ with an existing infrastructure. Any process can connect to

external processes, or it can be managed by the discovery service only. Every external process should have a type and name in the same way as TorQ processes, to enable them to be located and used as required.

Non-TorQ processes need to be listed by default in $KDBCONFIG/settings/nontorqprocess.csv. This file has the same format as the standard process.csv file. The location of the non-TorQ process file can be adjusted using the .servers.NONTORQPROCESSFILE variable. To enable connections, set .servers.TRACKNONTORQPROCESS to 1b.

Example of nontorqprocess.csv file:

```
host,port,proctype,procname
aquaq,5533,hdb,extproc01
aquaq,5577,hdb,extproc02
```

5.6 Manually Adding And Using Connections

Connections can also be manually added and used. See .api.p".servers.*" for details.

Chapter 6

Utilities

We have provided several utility scripts, which either implement developer aids or standard operations which are useful across processes.

6.1 api.q

This provides a mechanism for documenting and publishing function/variable/table or view definitions within the kdb+ process. It provides a search facility both by name and definition (in the case of functions). There is also a function for returning the approximate memory usage of each variable in the process in descending order.

Definitions are added using the .api.add function. A variable can be marked as public or private, and given a description, parameter list and return type. The search functions will return all the values found which match the pattern irrespective of them having a pre-defined definition.

Whether a value is public or private is defined in the definitions table. If not found then by default all values are private, except those which live in the .q or top level namespace.

.api.f is used to find a function, variable, table or view based on a case-insensitive pattern search. If a symbol parameter is supplied, a wildcard search of *[suppliedvalue]* is done. If a string is supplied, the value is used as is, meaning other non-wildcard regex pattern matching can be done.

```
q).api.f`max
name                | vartype   namespace public descrip             ..
--------------------| ------------------------------------------------..
maxs                | function  .q        1      ""                  ..
mmax                | function  .q        1      ""                  ..
.clients.MAXIDLE    | variable  .clients  0      ""                  ..
.access.MAXSIZE     | variable  .access   0      ""                  ..
.cache.maxsize      | variable  .cache    1      "The maximum size in..
.cache.maxindividual| variable  .cache    1      "The maximum size in..
```

```
max                 | primitive           1      ""                    ..
q).api.f"max*"
name| vartype   namespace public descrip params return
----| ------------------------------------------------
maxs| function  .q        1      ""      ""     ""
max | primitive           1      ""      ""     ""
```

.api.p is the same as .api.f, but only returns public functions. .api.u is as .api.p, but only includes user defined values i.e. it excludes q primitives and values found in the .q, .Q, .h and .o namespaces. .api.find is a more general version of .api.f which can be used to do case sensitive searches.

.api.s is used to search function definitions for specific values.

```
q).api.s"*max*"
function            definition                                          ..
--------------------------------------------------------------------------..
.Q.w                "k){`used`heap`peak`wmax`mmap`mphy`syms`symw!(.\"..
.clients.cleanup    "{if[count w0:exec w from`.clients.clients where ..
.access.validsize   "{[x;y;z] $[superuser .z.u;x;MAXSIZE>s:-22!x;x;'\..
.servers.getservers "{[nameortype;lookups;req;autoopen;onlyone]\n r:$..
.cache.add          "{[function;id;status]\n \n res:value function;\n..
```

.api.m is used to return the approximate memory usage of variables and views in the process, retrieved using -22!. Views will be re-evaluated if required. Use .api.mem[0b] if you do not want to evaluate and return views.

```
q).api.m[]
variable          size    sizeMB
--------------------------------
.tz.t             1587359 2
.help.TXT         15409   0
.api.detail       10678   0
.proc.usage       3610    0
.proc.configusage 1029    0
..
```

.api.whereami[lambda] can be used to retrieve the name of a function given its definition. This can be useful in debugging.

```
q)g:{x+y}
q)f:{20 + g[x;10]}
q)f[10]
40
q)f[`a]
{x+y}
'type
+
`a
10
q)).api.whereami[.z.s]
`..g
```


6.2 timer.q

kdb+ provides a single timer function, .z.ts which is triggered with the frequency specified by -t. We have provided an extension to allow multiple functions to be added to the timer and fired when required. The basic concept is that timer functions are registered in a table, with .z.ts periodically checking the table and running whichever functions are required. This is not a suitable mechanism where very high frequency timers are required (e.g. sub 500ms).

There are two ways a function can be added to a timer- either as a repeating timer, or to fire at a specific time. When a repeating timer is specified, there are three options as to how the timer can be rescheduled. Assuming that a timer function with period P is scheduled to fire at time T0, actually fires at time T1 and finishes at time T2, then

- mode 0 will reschedule for T0+P;
- mode 1 will reschedule for T1+P;
- mode 2 will reschedule for T2+P.

Both mode 0 and mode 1 have the potential for causing the timer to back up if the finish time T2 is after the next schedule time. See .api.p".timer.*"for more details.

6.3 async.q

kdb+ processes can communicate with each using either synchronous or asynchronous calls. Synchronous calls expect a response and so the server must process the request when it is received to generate the result and return it to the waiting client. Asynchronous calls do not expect a response so allow for greater flexibility. The effect of synchronous calls can be replicated with asynchronous calls in one of two ways (further details in section 8.2):

- deferred synchronous: the client sends an async request, then blocks on the handle waiting for the result. This allows the server more flexibility as to how and when the query is processed;
- asynchronous postback: the client sends an async request which is wrapped in a function to be posted back to the client when the result is ready. This allows the server flexibility as to how and when the query is processed, and allows the client to continue processing while the server is generating the result.

The code for both of these can get a little tricky, largely due to the amount of error trapping required. We have provided two functions to allow these methods to be used more easily. .async.deferred takes a list of handles and a query, and will return a two item list of (success;results).


```
q).async.deferred[3 5;({system"sleep 1";system"p"};())]
1    1
9995 9996
q).async.deferred[3 5;({x+y};1;2)]
1 1
3 3
q).async.deferred[3 5;({x+y};1;`a)]
0                          0
"error: server fail:type" "error: server fail:type"
q).async.deferred[3 5 87;({system"sleep 1";system"p"};())]
1     1     0
9995i 9996i "error: comm fail: failed to send query"
```

.async.postback takes a list of handles, a query, and the name or lambda of the postback function to return the result to. It will immediately return a success vector, and the results will be posted back to the client when ready.

```
q).async.postback[3 5;({system"sleep 1";system"p"};());`showresult]
11b
q)
q)9995i
9996i

q).async.postback[3 5;({x+y};1;2);`showresult]
11b
q)3
3

q).async.postback[3 5;({x+y};1;`a);`showresult]
11b
q)"error: server fail:type"
"error: server fail:type"

q).async.postback[3 5;({x+y};1;`a);showresult]
11b
q)"error: server fail:type"
"error: server fail:type"

q).async.postback[3 5 87;({x+y};1;2);showresult]
110b
q)3
3
```

For more details, see .api.p".async.*".

6.4 cache.q

cache.q provides a mechanism for storing function results in a cache and returning them from the cache if they are available and non stale. This can greatly boost performance for frequently run queries.

The result set cache resides in memory and as such takes up space. It is up to the programmer to determine which functions are suitable for caching. Likely candidates are those where some or all of the following conditions hold:

- the function is run multiple times with the same parameters (perhaps different clients all want the same result set);
- the result set changes infrequently or the clients can accept slightly out-of-date values;
- the result set is not too large and/or is relatively expensive to produce. For example, it does not make sense to cache raw data extracts.

The cache has a maximum size and a minimum size for any individual result set, both of which are defined in the configuration file. Size checks are done with -22! which will give an approximation (but underestimate) of the result set size. In the worst case the estimate could be half the size of the actual size.

If a new result set is to be cached, the size is checked. Assuming it does not exceed the maximum individual size then it is placed in the cache. If the new cache size would exceed the maximum allowed space, other result sets are evicted from the cache. The current eviction policy is to remove the least recently accessed result sets until the required space is freed. The cache performance is tracked in a table. Cache adds, hits, fails, reruns and evictions are monitored.

The main function to use the cache is .cache.execute[function; staletime]. If the function has been executed within the last staletime, then the result is returned from the cache. Otherwise the function is executed and placed in the cache.

The function is run and the result placed in the cache:

```
q)\t r:.cache.execute[({system"sleep 2"; x+y};1;2);0D00:01]
2023
q)r
3
```

The second time round, the result set is returned immediately from the cache as we are within the staletime value:

```
q)\t r1:.cache.execute[({system"sleep 2"; x+y};1;2);0D00:01]
0
q)r1
3
```

If the time since the last execution is greater than the required stale time, the function is re-run, the cached result is updated, and the result returned:

```
q)\t r2:.cache.execute[({system"sleep 2"; x+y};1;2);0D00:00]
2008
q)r2
3
```

The cache performance is tracked:

```
q).cache.getperf[]
time                          id status function
-------------------------------------------------------------------
```



```
2013.11.06D12:41:53.103508000 2  add    {system"sleep 2"; x+y} 1 2
2013.11.06D12:42:01.647731000 2  hit    {system"sleep 2"; x+y} 1 2
2013.11.06D12:42:53.930404000 2  rerun  {system"sleep 2"; x+y} 1 2
```

See .api.p".cache.*"for more details.

6.5 email.q

A library file is provided to allow TorQ processes to send emails using an SMTP server. This is a wrapper around the standard libcurl library. The library file is currently available for Windows (32 bit), Linux (32 and 64 bit) and OSX (32 and 64 bit). The associated q script contains two main methods for creating a connection and sending emails. The email library requires a modification to the path to find the required libs - see the top of email.q for details.

The main connection method .email.connect takes a single dictionary parameter and returns 0i for success and -1i for failure.

Parameter	Req	Type	Description
url	Y	symbol	URL of mail server e.g. smtp://mail.example.com
user	Y	symbol	Username of user to login as
password	Y	symbol	Password for user
usessl	N	boolean	Connect using SSL/TLS, defaults to false
from	N	symbol	Email from field, defaults to torq@aquaq.co.uk
debug	N	integer	Debug level. 0=no output, 1=normal output, 2=verbose output. Default is 1

Table 6.1: Email connection creation parameters

An example is:

```
q).email.connect[`url`user`password`from`usessl`debug!(`$"smtp://mail.example.com
    :80";`$"torquser@aquaq.co.uk";`hello;`$"torquser@aquaq.co.uk";0b;1i)]
02 Jan 2015 11:45:19   emailConnect: url is set to smtp://mail.example.com:80
02 Jan 2015 11:45:19   emailConnect: user is set to torquser@aquaq.co.uk
02 Jan 2015 11:45:19   emailConnect: password is set
02 Jan 2015 11:45:19   emailConnect: from is set torquser@aquaq.co.uk
02 Jan 2015 11:45:19   emailConnect: trying to connect
02 Jan 2015 11:45:19   emailConnect: connected, socket is 5
0i
```

The email sending function .email.send takes a single dictionary parameter containing the details of the email to send. A connection must be established before an email can be sent. The send function returns an integer of the email length on success, or -1 on failure.

Parameter	Req	Type	Description
to	Y	symbol (list)	addresses to send to
subject	Y	char list	email subject
body	Y	list of char lists	email body
cc	N	symbol (list)	cc list
bodyType	N	symbol	type of email body. Can be ‘text or ‘html. Default is ‘text
attachment	N	symbol (list)	files to attach
image	N	symbol	image to add to bottom of email body, default is $KDBHTML/img/logo.jpg. Use ‘none for no image
debug	N	integer	Debug level. 0=no output, 1=normal output, 2=verbose output. Default is 1

Table 6.2: Email sending parameters

An example is:

```
q).email.send[`to`subject`body`debug!(`$"test@aquaq.co.uk";"test email";("hi";"this
    is an email from torq");1i)]
02 Jan 2015 12:39:29   sending email with subject: test email
02 Jan 2015 12:39:29   email size in bytes is 16682
02 Jan 2015 12:39:30   emailSend: email sent
16682i
```

Note that if emails are sent infrequently the library must re-establish the connection to the mail server (this will be done automatically after the initial connection). In some circumstances it may be better to batch emails together to send, or to offload email sending to separate processes as communication with the SMTP server can take a little time.

Two further functions are available, .email.connectdefault and .email.senddefault. These are as above but will use the default configuration defined within the configuration files as the relevant parameters passed to the methods. In addition, .email.senddefault will automatically establish a connection.

```
q).email.senddefault[`to`subject`body!(`$"test@aquaq.co.uk";"test email";("hi";"this
    is an email from torq"))]
2015.01.02D12:43:34.646336000|aquaq||discovery1|INF|email|sending email
2015.01.02D12:43:35.743887000|aquaq||discovery1|INF|email|connection to mail server
    successful
2015.01.02D12:43:37.250427000|aquaq|discovery1|INF|email|email sent
16673i
q).email.senddefault[`to`subject`body!(`$"test@aquaq.co.uk";"test email 2";("hi";"
    this is an email from torq"))]
2015.01.02D12:43:48.115403000|aquaq|discovery1|INF|email|sending email
2015.01.02D12:43:49.385807000|aquaq|discovery1|INF|email|email sent
16675i
```


.email.test will attempt to establish a connection to the default configured email server and send a test email to the specified address. debug should be set to 2i (verbose) to extract the full information.

```
q).email.debug:2i
q).email.test `$"test@aquaq.co.uk"
...
```

Additionally functions are available within the email library. See .api.p".email.*"for more details.

6.6 timezone.q

A slightly customised version of the timezone conversion functionality from code.kx[1]. It loads a table of timezone information from $KDBCONFIG. See .api.p".tz.*"for more details.

6.7 compress.q

compress.q applies compression to any kdb+ database, handles all partition types including date, month, year, int, and can deal with top level splayed tables. It will also decompress files as required. Once the compression/decompression is complete, summary statistics are returned, with detailed statistics for each compressed or decompressed file held in a table.

The utility is driven by the configuration specified within a csv file. Default parameters can be given, and these can be used to compress all files within the database. However, the compress.q utility also provides the flexibility to compress different tables with different compression parameters, and different columns within tables using different parameters. A function is provided which will return a table showing each file in the database to be compressed, and how, before the compression is performed.

Compression is performed using the -19! operator, which takes 3 parameters; the compression algorithm to use (0 - none, 1 - kdb+ IPC, 2 - gzip), the compression blocksize as a power of 2 (between 12 and 19), and the level of compression to apply (from 0 - 9, applicable only for gzip). (For further information on -19! and the parameters used, see code.kx.com[2].)

The compressionconfig.csv file should have the following format:

```
table,minage,column,calgo,cblocksize,clevel
default,20,default,2,17,6
trades,20,default,1,17,0
```

[1] http://code.kx.com/wiki/Cookbook/Timezones
[2] code.kx.com/wiki/Cookbook/FileCompression


```
quotes,20,asize,2,17,7
quotes,20,bsize,2,17,7
```

This file can be placed in the config folder, or a path to the file given at run time.

The compression utility compresses all tables and columns present in the HDB but not specified in the driver file according the default parameters. In effect, to compress an entire HDB using the same compression parameters, a single row with name default would suffice. To specify that a particular table should be compressed in a certain different manner, it should be listed in the table. If default is given as the column for this table, then all of the columns of that table will be compressed accordingly. To specify the compression parameters for particular columns, these should be listed individually. For example, the file above will compress trades tables 20 days old or more with an algorithm of 1, and a blocksize of 17. The asize and bsize columns of any quotes tables older than 20 days old will be compressed using algorithm 2, blocksize 17 and level 7. All other files present will be compressed according to the default, using an algorithm 2, blocksize 17 and compression level 6. To leave files uncompressed, you must specify them explicitly in the table with a calgo of 0. If the file is already compressed, note that an algorithm of 0 will decompress the file.

This utility should be used with caution. Before running the compression it is recommended to run the function .cmp.showcomp, which takes three parameters - the path to the database, the path to the csv file, and the maximum age of the files to be compressed:

```
.cmp.showcomp[`:/full/path/to/HDB;.cmp.inputcsv;maxage]
              /- for using the csv file in the config folder
.cmp.showcomp[`:/full/path/to/HDB;`:/full/path/to/csvfile;maxage]
              /- to specify a file
```

This function produces a table of the files to be compressed, the parameters with which they will be compressed, and the current size of the file. Note that the current size column is calculated using hcount; on a file which is already compressed this returns the uncompressed length, i.e. this cannot be used as a signal as to whether the file is compressed already.

```
fullpath                        column table  partition  age calgo cblocksize clevel
    compressage currentsize
-------------------------------------------------------------------------------------
:/home/hdb/2013.11.05/depth/asize1 asize1 depth  2013.11.05 146 0     17         8
         1           787960
:/home/hdb/2013.11.05/depth/asize2 asize2 depth  2013.11.05 146 0     17         8
         1           787960
:/home/hdb/2013.11.05/depth/asize3 asize3 depth  2013.11.05 146 0     17         8
         1           787960
:/home/hdb/2013.11.05/depth/ask1   ask1   depth  2013.11.05 146 0     17         8
         1           1575904
....
```

To then run the compression function, use .cmp.compressmaxage with the same parameters as .cmp.showcomp (hdb path, csv path, maximum age of files):


```
.cmp.compressmaxage[`:/full/path/to/HDB;.cmp.inputcsv;maxage]
                /- for using the csv file in the config folder
.cmp.compressmaxage[`:/full/path/to/HDB;`:/full/path/to/csvfile;maxage]
                /- to specify a file
```

To run compression on all files in the database disregarding the maximum age of the files (i.e. from minage as specified in the configuration file to infinitely old), then use:

```
.cmp.docompression[`:/full/path/to/HDB;.cmp.inputcsv]
                /- for using the csv file in the config folder
.cmp.docompression[`:/full/path/to/HDB;`:/full/path/to/csvfile]
                /- to specify a file
```

Logs are produced for each file which is compressed or decompressed. Once the utility is complete, the statistics of the compression are also logged. This includes the memory savings in MB from compression, the additional memory usage in MB for decompression, the total compression ratio, and the total decompression ratio:

```
|comp1|INF|compression|Memory savings from compression: 34.48MB. Total compression
    ratio: 2.51.
|comp1|INF|compression|Additional memory used from de-compression: 0.00MB. Total de-
    compression ratio: .
|comp1|INF|compression|Check .cmp.statstab for info on each file.
```

A table with the compressed and decompressed length for each individual file, in descending order of compression ratio, is also produced. This can be found in .cmp.statstab:

```
file                    algo compressedLength uncompressedLength compressionratio
-------------------------------------------------------------------------------
:/hdb/2014.03.05/depth/asize1 2    89057            772600             8.675343
:/hdb/2014.01.06/depth/asize1 2    114930           995532             8.662073
:/hdb/2014.03.05/depth/bsize1 2    89210            772600             8.660464
:/hdb/2014.03.12/depth/bsize1 2    84416            730928             8.658643
:/hdb/2014.01.06/depth/bsize1 2    115067           995532             8.651759
.....
```

A note for windows users - windows supports compression only with a compression blocksize of 16 or more.

6.8 dataloader.q

This script contains some utility functions to assist in loading data from delimited files (e.g. comma separated, tab delimited). It is a more generic version of the data loader example on code.kx[3]. The supplied functions allow data to be read in configurable size chunks and written out to the database. When all the data is written, the on-disk data is re-sorted and the attributes are applied. The main function is .loader.loadalldata which

[3] http://code.kx.com/wiki/Cookbook/LoadingFromLargeFiles

takes two parameters- a dictionary of loading parameters and a directory containing the files to read. The dictionary should/can have the following fields:

Parameter	Req	Type	Description
headers	Y	symbol list	Names of the header columns in the file
types	Y	char list	Data types to read from the file
separator	Y	char[list]	Delimiting character. Enlist it if first line of file is header data
tablename	Y	symbol	Name of table to write data to
dbdir	Y	symbol	Directory to write data to
partitiontype	N	symbol	Partitioning to use. Must be one of ‘date‘month‘year‘int. Default is ‘date
partitioncol	N	symbol	Column to use to extract partition information. Default is ‘time
dataprocessfunc	N	function	Diadic function to process data after it has been read in. First argument is load parameters dictionary, second argument is data which has been read in. Default is {[x;y] y}
chunksize	N	int	Data size in bytes to read in one chunk. Default is 100 MB
compression	N	int list	Compression parameters to use e.g. 17 2 6. Default is empty list for no compression
gc	N	boolean	Whether to run garbage collection at appropriate points. Default is 0b (false)

Table 6.3: Data loader function parameters

Example usage:

```
.loader.loadallfiles[`headers`types`separator`tablename`dbdir!(`sym`time`price`volume
    ;"SP  FI";",";`trade;`:hdb); `:TDC/toload]
.loader.loadallfiles[`headers`types`separator`tablename`dbdir`dataprocessfunc`
    chunksize`partitiontype`partitioncol`compression`gc!(`sym`time`price`volume;"SP
    FI";enlist",";`tradesummary;`:hdb;{[p;t] select sum size, max price by date:time.
    date from t};`int$500*2 xexp 20;`month;`date;16 1 0;1b); `:TDC/toload]
```

6.9 subscriptions.q

The subscription utilities allow multiple subscriptions to different data sources to be managed and maintained. Automatic resubscriptions in the event of failure are possible, along as specifying whether the process will get the schema and replay the log file from the remote source (e.g. in the case of tickerplant subscriptions).

.sub.getsubscriptionhandles is used to get a table of processes to subscribe to. The following can be used to return a table of all connected processes of type tickerplant:


```
.sub.getsubscriptionhandles[`tickerplant;`;()!()]
```

.sub.subscribe is used to subscribe to a process for the supplied list of tables and instruments. For example, to subscribe to instruments A, B and C for the quote table from all tickerplants:

```
.sub.subscribe[`trthquote;`A`B;0b;0b] each .sub.getsubscriptionhandles[`tickerplant
    ;`;()!()]
```

The subscription method uses backtick for "all" (which is the same as kdb+tick). To subscribe to all tables, all instruments, from all tickerplants:

```
.sub.subscribe[`;`;0b;0b] each .sub.getsubscriptionhandles[`tickerplant;`;()!()]
```

See .api.p".sub.*" for more details.

6.10 pubsub.q

pubsub.q is essentially a placeholder script to allow publish and subscribe functionality to be implemented. Licenced kdb+tick users can use the publish and subscribe functionality implemented in u.[k|q]. If u.[k|q] is placed in the common code directory and loaded before pubsub.q (make sure u.[k|q] is listed before pubsub.q in order.txt) then publish and subscribe will be implemented. You can also build out this file to add your own publish and subscribe routines as required.

6.11 tplogutils.q

tplogutils.q contains functions for recovering tickerplant log files. Under certain circumstances the tickerplant log file can become corrupt by having an invalid sequence of bytes written to it. A log file can be recovered using a simple recovery method[4]. However, this will only recover messages up to the first invalid message. The recovery functions defined in tplogutils.q allow all valid messages to be recovered from the tickerplant log file.

6.12 monitoringchecks.q

monitoringchecks.q implements a set of standard, basic monitoring checks. They include checks to ensure:

- table sizes are increasing during live capture

[4] http://code.kx.com/wsvn/code/contrib/simon/tickrecover/rescuelog.q

- the HDB data saves down correctly
- the allocated memory of a process does not increase past a certain size
- the size of the symbol list in memory doesn't grow to big
- the process does not have too much on its pending subscriber queue

These checks are intended to be run by the reporter process on a schedule, and any alerts emailed to an appropriate recipient list.

6.13 heartbeat.q

heartbeat.q implements heartbeating, and relies on both timer.q and pubsub.q. A table called heartbeat will be published periodically, allowing downstream processes to detect the availability of upstream components. The heartbeat table contains a heartbeat time and counter. The heartbeat script contains functions to handle and process heartbeats and manage upstream process failures. See .api.p".hb.*"for details.

6.14 dbwriteutils.q

This contains a set of utility functions for writing data to historic databases.

6.14.1 Sorting and Attributes

The sort utilities allow the sort order and attributes of tables to be globally defined. This helps to manage the code base when the data can potentially be written from multiple locations (e.g. written from the RDB, loaded from flat file, replayed from the tickerplant log). The configuration is defined in a csv which defaults to $KDBCONFG/sort.csv. The default setup is that every table is sorted by sym and time, with a p attribute on sym (this is the standard kdb+ tick configuration).

```
aquaq$ tail config/sort.csv
tabname,att,column,sort
default,p,sym,1
default,,time,1
```

As an example, assume we have an optiontrade table which we want to be different from the standard set up. We would like the table to be sorted by optionticker and then time, with a p attribute on optionticker. We also have a column called underlyingticker which we can put an attribute on as it is derived from optionticker (so there is an element of de-normalisation present in the table). We also have an exchange field which we would like to put a g attribute on. All other tables we want to be sorted and parted in the

standard way. The configuration file would look like this (sort order is derived from the order within the file combined with the sort flag being set to true):

```
aquaq$ tail config/sort.csv
tabname,att,column,sort
default,p,sym,1
default,,time,1
optiontrade,p,optionticker,1
optiontrade,,exchtime,1
optiontrade,p,underlyingticker,0
optiontrade,g,exchange,0
```

To invoke the sort utilities, supply a list of (tablename; partitions) e.g.

```
q).sort.sorttab(`trthtrade;`:hdb/2014.11.20/trthtrade`:hdb/2014.11.20/trthtrade)
2014.12.03D09:56:19.214006000|aquaq|test|INF|sort|sorting the trthtrade table
2014.12.03D09:56:19.214045000|aquaq|test|INF|sorttab|No sort parameters have been
    specified for : trthtrade. Using default parameters
2014.12.03D09:56:19.214057000|aquaq|test|INF|sortfunction|sorting :hdb/2014.11.19/
    trthtrade/ by these columns : sym, time
2014.12.03D09:56:19.219716000|aquaq|test|INF|applyattr|applying p attr to the sym
    column in :hdb/2014.11.19/trthtrade/
2014.12.03D09:56:19.220846000|aquaq|test|INF|sortfunction|sorting :hdb/2014.11.20/
    trthtrade/ by these columns : sym, time
2014.12.03D09:56:19.226008000|aquaq|test|INF|applyattr|applying p attr to the sym
    column in :hdb/2014.11.20/trthtrade/
2014.12.03D09:56:19.226636000|aquaq|test|INF|sort|finished sorting the trthtrade
    table
```

A different sort configuration file can be loaded with

```
.sort.getsortcsv[`:file]
```

6.14.2 Garbage Collection

The garbage collection utility prints some debug information before and after the garbage collection.

```
q).gc.run[]
2014.12.03D10:22:51.688435000|aquaq|test|INF|garbagecollect|Starting garbage collect.
     mem stats: used=2 MB; heap=1984 MB; peak=1984 MB; wmax=0 MB; mmap=0 MB; mphy
    =16384 MB; syms=0 MB; symw=0 MB
2014.12.03D10:22:53.920656000|aquaq|test|INF|garbagecollect|Garbage collection
    returned 1472MB. mem stats: used=2 MB; heap=512 MB; peak=1984 MB; wmax=0 MB; mmap
    =0 MB; mphy=16384 MB; syms=0 MB; symw=0 MB
```

6.14.3 Table Manipulation

The table manipulation utilities allow table manipulation routines to be defined in a single place. This is useful when data can be written from mutliple different processes e.g. RDB, WDB, or tickerplant log replay. Instead of having to create a separate definition of customised manipulation in each process, it can be done in a single location and invokved in each process.

6.15 help.q

The standard help.q from code.kx provides help utilities in the console. This should be kept up to date with code.kx[5].

```
q)help`
adverb    | adverbs/operators
attributes| data attributes
cmdline   | command line parameters
data      | data types
define    | assign, define, control and debug
dotz      | .z locale contents
errors    | error messages
save      | save/load tables
syscmd    | system commands
temporal  | temporal - date & time casts
verbs     | verbs/functions
```

6.16 html.q

An HTML utility has been added to accompany the HTML5 front end for the Monitoring process. It includes functions to format dates, tables to csv to configure the HTML file to work on the correct process. It is accessible from the `.html` namespace.

6.17 Additional Utilities

There are some additional user contributed utility scripts available on code.kx which are good candidates for inclusion. These could either be dropped into the common code directory, or if not globally applicable then in the code directory for either the process type or name. The full set of user contributed code is documented here[6].

6.18 Full API

The full public api can be found by running

```
q).api.u`
name              | vartype  namespace public descrip                 ..
------------------| ------------------------------------------------..
.proc.createlog   | function .proc     1      "Create the standard out..
.proc.rolllogauto| function .proc     1      "Roll the standard out/e..
.proc.loadf       | function .proc     1      "Load the specified file..
.proc.loaddir     | function .proc     1      "Load all the .q and .k ..
.lg.o             | function .lg       1      "Log to standard out"   ..
..
```

[5] http://code.kx.com/wsvn/code/kx/kdb+/d/help.q
[6] http://code.kx.com/wiki/Contrib

Combined with the commented configuration file, this should give a good overview of the functionality available. A description of the individual namespaces is below- run .api.u "namespace*"to list the functions.

Namespace	Description
.proc	Process API
.lg	Standard out/error logging API
.err	Error throwing API
.usage	Usage logging API
.access	Permissions API
.clients	Client tracking API
.servers	Server tracking API
.async	Async communication API
.timer	Timer API
.cache	Caching API
.tz	Timezone conversions API
.checks	Monitoring API
.cmp	Compression API
.ps	Publish and Subscribe API
.hb	Heartbeating API
.loader	Data Loader API
.sort	Data sorting and attribute setting API
.sub	Subscription API
.gc	Garbage Collection API
.tplog	Tickerplant Log Replay API
.api	API management API

Table 6.4: Full API

Chapter 7

Visualisation

kdb+ supports websockets and so HTML5 GUIs can be built. We have incorporated a set of server side and client side utilities to ease HTML GUI development.

7.1 kdb+ Utilities

The server side utilities are contained in html.q. These utilise some community code, specifically json.k and a modified version of u.q, both from Kx Systems. The supplied functionality includes:

- json.k provides two way conversion between kdb+ data structures and JSON;
- u.q is the standard pub/sub functionality provided with kdb+tick, and a modified version is incorporated to publish data structures which can be easily interpreted in JavaScript;
- functions for reformatting temporal types to be JSON compliant;
- page serving to utilise the inbuilt kdb+ webserver to serve custom web pages. An example would be instead of having to serve a page with a hardcoded websocket connection host and port, the kdb+ process can serve a page connecting back to itself no matter which host or port it is running on.

7.2 JavaScript Utilities

The JavaScript utilities are contained in kdbconnect.js. The library allows you to:

- create a connection to the kdb+ process;
- display the socket status;

- sending queries;
- binding results returned from kdb+ to updates in the webpage.

7.3 Outline

All communication between websockets and kdb+ is asynchronous. The approach we have adopted is to ensure that all data sent to the web browser is encoded as a JSON object containing a tag to enable the web page to decipher what the data relates to. The format we have chosen is for kdb+ to send dictionaries of the form:

```
`name`data!("dataID";dataObject)
```

All the packing can be done by .html.dataformat. Please note that the temporal types are converted to longs which can easily be converted to JavaScript Date types. This formatting can be modified in the formating dictionary .html.typemap.

```
q)a:flip `minute`time`date`month`timestamp`timespan`datetime`float`sym!enlist each
    (09:00; 09:00:00.0;.z.d; `month$.z.d; .z.p; .z.n;.z.z;20f;`a)
q).html.dataformat["start";(enlist `tradegraph)!enlist a]
name| "start"
data| (,`tradegraph)!,+`minute`time`date`month`timestamp`timespan`datetime`float`sym
    !(,32400000;,32400000;,1396828800000;,1396310400000;,"2014-04-07T13:23:01Z
    ";,48181023;,"2014-04-07T13:23:01Z";,20f;,`a)
q)first (.html.dataformat["start";(enlist `tradegraph)!enlist a])[`data;`tradegraph]
minute   | 32400000
time     | 32400000
date     | 1396828800000
month    | 1396310400000
timestamp| "2014-04-07T13:23:01Z"
timespan | 48181023
datetime | "2014-04-07T13:23:01Z"
float    | 20f
sym      | `a
```

We have also extended this structure to allow web pages to receive data in a way similar to the standard kdb+tick pub/sub format. In this case, the data object looks like:

```
`name`data!("upd";`tablename`tabledata!(`trade;([]time:09:00 09:05 09:10; price:12 13
     14)))
```

This can be packed with .html.updformat:

```
q).html.updformat["upd";`tablename`tabledata!(`trade;a)]
name| "upd"
data| `tablename`tabledata!(`trade;+`minute`time`date`month`timestamp`timespan`
    datetime`float`sym
    !(,32400000;,32400000;,1396828800000;,1396310400000;,"2014-04-07T13:23:01Z
    ";,48181023;,"2014-04-07T13:23:01Z";,20f;,`a))
q)first(.html.updformat["upd";`tablename`tabledata!(`trade;a)])[`data;`tabledata]
minute   | 32400000
time     | 32400000
```



```
date     | 1396828800000
month    | 1396310400000
timestamp| "2014-04-07T13:23:01Z"
timespan | 48181023
datetime | "2014-04-07T13:23:01Z"
float    | 20f
sym      | `a
```

To utilise the pub/sub functionality, the web page must connect to the kdb+ process and subscribe for updates. Subscriptions are done using

```
.html.wssub[`tablename]
```

Publications from the kdb+ side are done with

```
.html.pub[`tablename;tabledata]
```

On the JavaScript side the incoming messages (data events) must be bound to page updates. For example, there might be an initialisation event called "start" which allows the web page to retrieve all the initial data from the process. The code below redraws the areas of the page with the received data.

```
/* Bind data - Data type "start" will execute the callback function */
KDBCONNECT.bind("data","start",function(data){
  // Check that data is not empty
  if(data.hbtable.length !== 0)
   // Write HTML table to div element with id heartbeat-table
   { $("#heartbeat-table").html(MONITOR.jsonTable(data.hbtable));}
  if(data.lmtable.length !== 0)
   // Write HTML table to div element with id logmsg-table
   { $("#logmsg-table").html(MONITOR.jsonTable(data.lmtable));}
  if(data.lmchart.length !== 0)
   // Log message error chart
   { MONITOR.barChart(data.lmchart,"logmsg-chart","Error Count","myTab"); }
  });
```

Similarly the upd messages must be bound to page updates. In this case, the structure is slightly different:

```
KDBCONNECT.bind("data","upd",function(data){
  if(data.tabledata.length===0) return;
  if(data.tablename === "heartbeat")
    { $("#heartbeat-table").html(MONITOR.jsonTable(data.tabledata));}
  if(data.tablename === "logmsg")
    { $("#logmsg-table").html(MONITOR.jsonTable(data.tabledata));}
  if(data.tablename === "lmchart")
    { MONITOR.barChart(data.tabledata,"logmsg-chart","Error Count","myTab"); }
 });
```

To display the WebSocket connection status the event "ws_event" must be bound and it will output one of these default messages: "Connecting...", "Connected" and "Disconnected" depending on the connection state of the WebSocket. Alternatively the value of the readyState attribute will determine the WebSocket status.


```
// Select html element using jQuery
var $statusMsg = $("#status-msg");
KDBCONNECT.bind("ws_event",function(data){
  // Data is the default message string
  $statusMsg.html(data);
});
KDBCONNECT.core.websocket.readyState // Returns 1 if connected.
```

Errors can be displayed by binding the event called "error".

```
KDBCONNECT.bind("error",function(data){
  $statusMsg.html("Error - " + data);
});
```

7.4 Example

A basic example is provided with the Monitor process. To get this to work, u.q from kdb+tick should be placed in the code/common directory to allow all processes to publish updates. It should be noted that this is not intended as a production monitoring visualisation screen, moreso a demonstration of functionality. See section 8.9.1 for more details.

7.5 Further Work

Further work planned includes:

- allow subscriptions on a key basis- currently all subscribers receive all updates;
- add JavaScript controls to allow in-place updates based on key pairs, and scrolling window updates e.g. add N new rows to top/bottom of the specified table;
- allow multiple websocket connections to be maintained at the same time.

Chapter 8

Processes

A set of processes is included. These processes build upon AquaQ TorQ, providing specific functionality. All the process scripts are contained in $KDBCODE/processes. All processes should have an entry in $KDBCONFIG/process.csv. All processes can have any type and name, except for discovery services which must have a process type of "discovery". An example process.csv is:

```
aquaq$ cat config/process.csv
host,port,proctype,procname
aquaq,9998,rdb,rdb_europe_1
aquaq,9997,hdb,rdb_europe_1aquaq,9999,hdb,hdb1
aquaq,9996,discovery,discovery1
aquaq,9995,discovery,discovery2
aquaq,8000,gateway,gateway1
aquaq,5010,tickerplant,tickerplant1
aquaq,5011,rdb,rdb1
aquaq,5012,hdb,hdb1
aquaq,5013,hdb,hdb2
aquaq,9990,tickerlogreplay,tpreplay1
aquaq,20000,kill,killhdbs
aquaq,20001,monitor,monitor1
aquaq,20002,housekeeping,hk1
```

8.1 Discovery Service

8.1.1 Overview

Processes use the discovery service to register their own availability, find other processes (by process type) and subscribe to receive updates for new process availability (by process type). The discovery service does not manage connections- it simply returns tables of registered processes, irrespective of their current availability. It is up to each individual process to manage its own connections.

The discovery service uses the process.csv file to make connections to processes on start up. After start up it is up to each individual process to attempt connections and register with the discovery service. This is done automatically, depending on the configuration parameters. Multiple discovery services can be run in which case each process will try to register and retrieve process details from each discovery process it finds in its process.csv file. Discovery services do not replicate between themselves. A discovery process must have its process type listed as discovery.

To run the discovery service, use a start line such as:

```
aquaq $ q torq.q -load code/processes/discovery.q -p 9995
```

Modify the configuration as required.

8.1.2 Operation

1. Processes register with the discovery service.

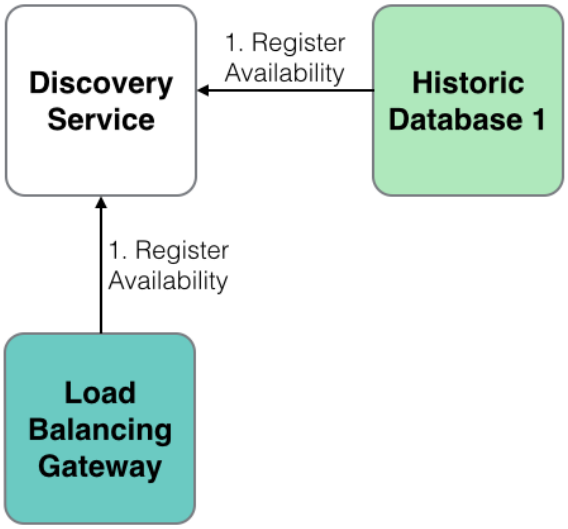

Figure 8.1: Discovery service registration

2. Processes use the discovery service to locate other processes.

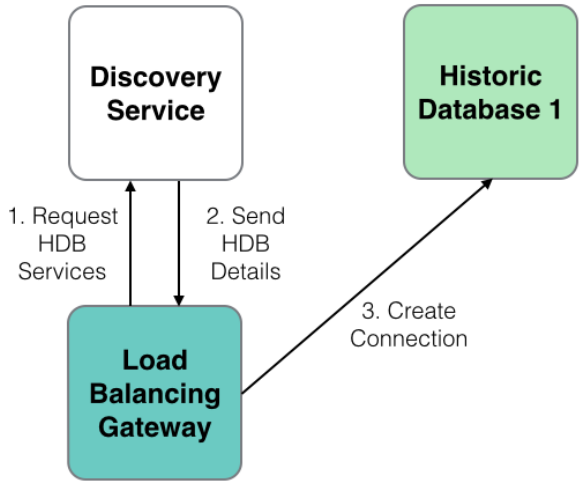

Figure 8.2: Process location

3. When new services register, any processes which have registered an interest in that process type are notified.

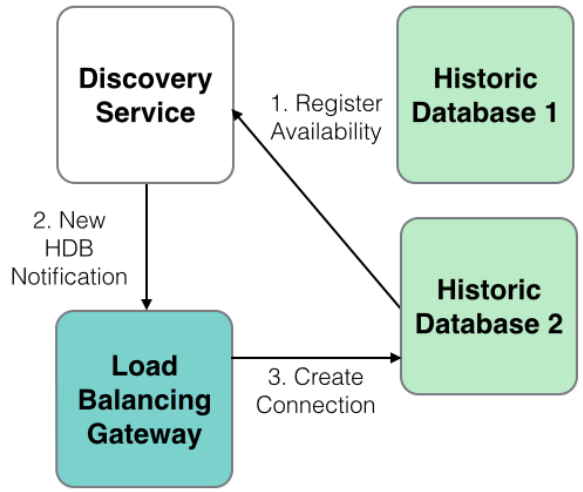

Figure 8.3: Notification

8.1.3 Available Processes

The list of available processes can be found in the .servers.SERVERS table.

```
q).servers.SERVERS
procname     proctype        hpup            w  hits startp
    lastp                        endp attributes
```



```
-------------------------------------------------------------------------------
discovery1   discovery       :aquaq:9995     0
    2014.01.22D17:00:40.947470000      ()!()
discovery2   discovery       :aquaq:9996     0
    2014.01.22D17:00:40.947517000      ()!()
hdb2         hdb             :aquaq:5013     0
    2014.01.22D17:00:40.947602000      ()!()
killtick     kill            :aquaq:20000    0
    2014.01.22D17:00:40.947602000      ()!()
tpreplay1    tickerlogreplay :aquaq:20002    0
    2014.01.22D17:00:40.947602000      ()!()
tickerplant1 tickerplant     :aquaq:5010  6  0    2014.01.22D17:00:40.967699000
    2014.01.22D17:00:40.967698000      ()!()
monitor1     monitor         :aquaq:20001 9  0    2014.01.22D17:00:40.971344000
    2014.01.22D17:00:40.971344000      ()!()
rdb1         rdb             :aquaq:5011  7  0    2014.01.22D17:06:13.032883000
    2014.01.22D17:06:13.032883000      `date`tables!(,2014.01.22;`fxquotes`heartbeat`
    logmsg`quotes`trades)
hdb3         hdb             :aquaq:5012  8  0    2014.01.22D17:06:18.647349000
    2014.01.22D17:06:18.647349000      `date`tables!(2014.01.13 2014.01.14;`fxquotes`
    heartbeat`logmsg`quotes`trades)
gateway1     gateway         :aquaq:5020  10 0    2014.01.22D17:06:32.152836000
    2014.01.22D17:06:32.152836000      ()!()
```

8.2 Gateway

A synchronous and asynchronous gateway is provided. The gateway can be used for load balancing and/or to join the results of queries across heterogeneous servers (e.g. an RDB and HDB). Ideally the gateway should only be used with asynchronous calls. Synchronous calls cause the gateway to block so limits the gateway to serving one query at a time (although if querying across multiple backend servers the backend queries will be run in parallel). When using asynchronous calls the client can either block and wait for the result (deferred synchronous) or post a call back function which the gateway will call back to the client with. With both asynchronous and synchronous queries the backend servers to execute queries against are selected using process type. The gateway API can be seen by querying .api.p".gw.*" within a gateway process.

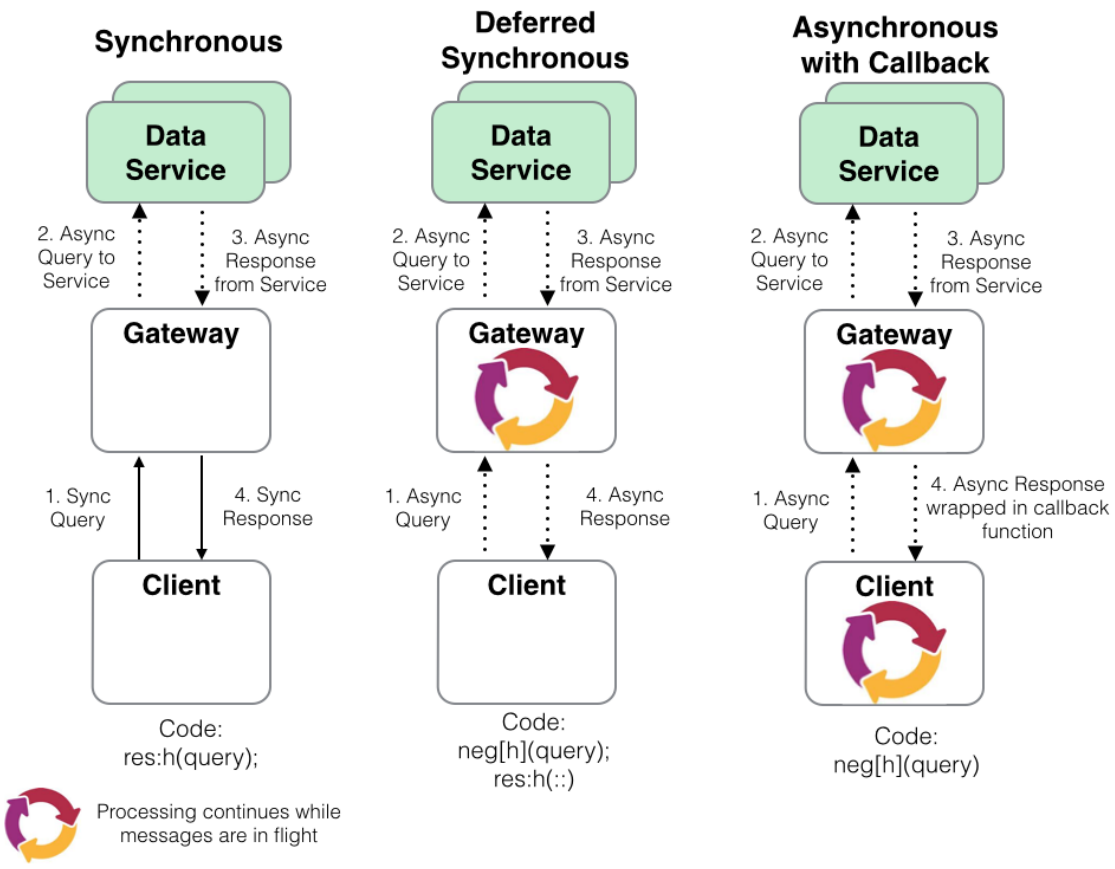

Figure 8.4: Gateway behaviour

8.2.1 Asynchronous Behaviour

Asynchronous queries allow much greater flexibility. They allow multiple queries to be serviced at once, prioritisation, and queries to be timed out. When an asynchronous query is received the following happens:

- the query is placed in a queue;
- the list of available servers is retrieved;
- the queue is prioritised, so those queries with higher priority are serviced first;
- queries are sent to back end servers as they become available. Once the backend server returns its result, it is given another query;
- when all the partial results from the query are returned the results are aggregated and returned to the client. They are either returned directly, or wrapped in a callback and posted back asynchronously to the client.

The two main customisable features of the gateway are the selection of available servers (.gw.availableservers) and the queue prioritisation (.gw.getnextqueryid). With default configuration, the available servers are those servers which are not currently servicing a query from the gateway, and the queue priority is a simple FIFO queue. The available servers could be extended to handle process attributes, such as the available datasets or the location of the process, and the queue prioritisation could be modified to anything required e.g. based on the query itself, the username, host of the client etc.

An asynchronous query can be timed out using a timeout defined by the client. The gateway will periodically check if any client queries have not completed in the alotted time, and return a timeout error to the client. If the query is already running on any backend servers then they cannot be timed out other than by using the standard -T flag.

8.2.2 Synchronous Behaviour

When using synchronous queries the gateway can only handle one query at a time and cannot timeout queries other than with the standard -T flag. All synchronous queries will be immediately dispatched to the back end processes. They will be dispatched using an asyhcnronous call, allowing them to run in parallel rather than serially. When the results are received they are aggregated and returned to the client.

8.2.3 Process Discovery

The gateway uses the discovery service to locate processes to query across. The discovery service will notify the gateway when new processes become available and the gateway will automatically connect and start using them. The gateway can also use the static information in process.csv, but this limits the gateway to a predefined list of processes rather than allowing new services to come online as demand requires.

8.2.4 Error Handling

When synchronous calls are used, q errors are returned to clients as they are encountered. When using asynchronous calls there is no way to return actual errors and appropriately prefixed strings must be used instead. It is up to the client to check the type of the received result and if it is a string then whether it contains the error prefix. The error prefix can be changed, but the default is "error: ". Errors will be returned when:

- the client requests a query against a server type which the gateway does not currently have any active instances of (this error is returned immediately);
- the query is timed out;

- a back end server returns an error;
- a back end server fails;
- the join function fails.

If postback functions are used, the error string will be posted back within the postback function (i.e. it will be packed the same way as a valid result).

8.2.5 Client Calls

There are four main client calls. The .gw.sync* methods should only be invoked synchronously, and the .gw.async* methods should only be invoked asynchronously. Each of these are documented more extensively in the gateway api. Use .api.p".gw.*" for more details.

Function	Description
.gw.syncexec[query; servertypes]	Execute the specified query synchronously against the required list of servers. If more than one server, the results will be razed.
.gw.syncexecj[query; servertypes; joinfunction]	Execute the specified query against the required list of servers. Use the specified join function to aggregate the results.
.gw.asyncexec[query; servertypes]	Execute the specified query against the required list of servers. If more than one server, the results will be razed. The client must block and wait for the results.
.gw.asyncexecjpt[query; servertypes; joinfunction; postback; timeout]	Execute the specified query against the required list of servers. Use the specified join function to aggregate the results. If the postback function is not set, the client must block and wait for the results. If it is set, the result will be wrapped in the specified postback function and returned asynchronously to the client. The query will be timed out if the timeout value is exceeded.

Table 8.1: Gateway API

For the purposes of demonstration, assume that the queries must be run across an RDB and HDB process, and the gateway has one RDB and two HDB processes available to it.

```
q).gw.servers
handle| servertype inuse active querycount lastquery                    usage
                  attributes
------| ---------------------------------------------------------------------
7     | rdb        0     1      17         2014.01.07D17:05:03.113927000 0D00
    :00:52.149069000 `datacentre`country!`essex`uk
```



```
8     | hdb        0     1      17         2014.01.07D17:05:03.113927000 0D00
    :01:26.143564000 `datacentre`country!`essex`uk
9     | hdb        0     1      2          2014.01.07D16:47:33.615538000 0D00
    :00:08.019862000 `datacentre`country!`essex`uk
12    | rdb        0     1      2          2014.01.07D16:47:33.615538000 0D00
    :00:04.018349000 `datacentre`country!`essex`uk
```

Both the RDB and HDB processes have a function f and table t defined. f will run for 2 seconds longer on the HDB processes then it will the RDB.

```
q)f
{system"sleep ",string x+$[`hdb=.proc.proctype;2;0]; t}
q)t
a
----
5013
5014
5015
5016
5017
```

Run the gateway. The main parameter which should be set is the .servers.CONNECTIONS parameter, which dictates the process types the gateway queries across. Also, we need to explicitly allow sync calls. We can do this from the config or from the command line.

```
q torq.q -load code/processes/gateway.q -p 8000 -.gw.synccallsallowed 1 -.servers.
    CONNECTIONS hdb rdb
```

Start a client and connect to the gateway. Start with a sync query. The HDB query should take 4 seconds and the RDB query should take 2 seconds. If the queries run in parallel, the total query time should be 4 seconds.

```
q)h:hopen 8000
q)h(`.gw.syncexec;(`f;2);`hdb`rdb)
a
----
5014
5015
5016
5017
5018
5012
5013
5014
5015
5016
q)\t h(`.gw.syncexec;(`f;2);`hdb`rdb)
4009
```

If a query is done for a server type which is not registered, an error is returned:

```
q)\t h(`.gw.syncexec;(`f;2);`hdb`rdb`other)
'not all of the requested server types are available; missing other
```

Custom join functions can be specified:


```
q)h(`.gw.syncexecj;(`f;2);`hdb`rdb;{sum{select count i by a from x} each x})
a   | x
----| -
5014| 2
5015| 2
5016| 2
5017| 1
5018| 1
5012| 1
5013| 1
```

Custom joins can fail with appropriate errors:

```
q)h(`.gw.syncexecj;(`f;2);`hdb`rdb;{sum{select count i by b from x} each x})
'failed to apply supplied join function to results: b
```

Asynchronous queries must be sent in async and blocked:

```
q)(neg h)(`.gw.asyncexec;(`f;2);`hdb`rdb); r:h(::)
        /- This white space is from pressing return
        /- the client is blocked and unresponsive

q)q)q)
q)
q)r
a
----
5014
5015
5016
5017
5018
5012
5013
5014
5015
5016
q)
```

We can send multiple async queries at once. Given the gateway has two RDBs and two HDBs avaialble to it, it should be possible to service two of these queries at the same time.

```
q)h:hopen each 8000 8000
q)\t (neg h)@\:(`.gw.asyncexec;(`f;2);`hdb`rdb); (neg h)@\:(::); r:h@\:(::)
4012
q)r
+(,`a)!,5014 5015 5016 5017 5018 5012 5013 5014 5015 5016
+(,`a)!,5013 5014 5015 5016 5017 9999 10000 10001 10002 10003
```

Alternatively async queries can specify a postback so the client does not have to block and wait for the result. The postback function must take two parameters- the first is the function that was sent up, the second is the results. The postback can either be a lambda, or the name of a function.

```
q)h:hopen 8000
```



```
q)handleresults:{-1(string .z.z)," got results"; -3!x; show y}
q)(neg h)(`.gw.asyncexecjpt;(`f;2);`hdb`rdb;raze;handleresults;0Wn)
q)
q)      /- These q prompts are from pressing enter
q)      /- The q client is not blocked, unlike the previous example
q)
q)2014.01.07T16:53:42.481 got results
a
----
5014
5015
5016
5017
5018
5012
5013
5014
5015
5016

/- Can also use a named function rather than a lambda
q)(neg h)(`.gw.asyncexecjpt;(`f;2);`hdb`rdb;raze;`handleresults;0Wn)
q)
q)
q)2014.01.07T16:55:12.235 got results
a
----
5014
5015
5016
5017
5018
5012
5013
5014
5015
5016
```

Asynchronous queries can also be timed out. This query will run for 22 seconds, but should be timed out after 5 seconds. There is a tolerance of +5 seconds on the timeout value, as that is how often the query list is checked. This can be reduced as required.

```
q)(neg h)(`.gw.asyncexecjpt;(`f;20);`hdb`rdb;raze;();0D00:00:05); r:h(::)


q)q)q)r
"error: query has exceeded specified timeout value"
q)\t (neg h)(`.gw.asyncexecjpt;(`f;20);`hdb`rdb;raze;();0D00:00:05); r:h(::)
6550
```

8.2.6 Non kdb+ Clients

All the examples in the previous section are from clients written in q. However it should be possible to do most of the above from non kdb+ clients. The officially supported

APIs for Java, C# and C allow the asynchronous methods above. For example, we can modify the try block in the main function of the Java Grid Viewer[1]:

```
import java.awt.BorderLayout;
import java.awt.Color;
import java.io.IOException;
import java.lang.reflect.Array;
import java.util.logging.Level;
import java.util.logging.Logger;
import javax.swing.JFrame;
import javax.swing.JScrollPane;
import javax.swing.JTable;
import javax.swing.table.AbstractTableModel;
import kx.c;

public class Main {
    public static class KxTableModel extends AbstractTableModel {
        private c.Flip flip;
        public void setFlip(c.Flip data) {
            this.flip = data;
        }

        public int getRowCount() {
            return Array.getLength(flip.y[0]);
        }

        public int getColumnCount() {
            return flip.y.length;
        }

        public Object getValueAt(int rowIndex, int columnIndex) {
            return c.at(flip.y[columnIndex], rowIndex);
        }

        public String getColumnName(int columnIndex) {
            return flip.x[columnIndex];
        }
    };

    public static void main(String[] args) {
        KxTableModel model = new KxTableModel();
        c c = null;
        try {
            c = new c("localhost", 8000,"username:password");
            // Create the query to send
            String query=".gw.asyncexec[(`f;2);`hdb`rdb]";
            // Send the query
            c.ks(query);
            // Block on the socket and wait for the result
            model.setFlip((c.Flip) c.k());
        } catch (Exception ex) {
            Logger.getLogger(Main.class.getName()).log(Level.SEVERE, null, ex);
        } finally {
            if (c != null) {try{c.close();} catch (IOException ex) {}
          }
        }
        JTable table = new JTable(model);
        table.setGridColor(Color.BLACK);
        String title = "kdb+ Example - "+model.getRowCount()+" Rows";
        JFrame frame = new JFrame(title);
```

[1] http://code.kx.com/wiki/Cookbook/InterfacingWithJava


```
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.getContentPane().add(new JScrollPane(table), BorderLayout.CENTER);
        frame.setSize(300, 300);
        frame.setVisible(true);
    }
}
```

Some of the unofficially supported APIs may only allow synchronous calls to be made.

8.3 Real Time Database (RDB)

The Real Time Database is a modified version of r.q found in kdb+tick. The modifications from the standard r.q include:

- Tickerplant (data source) and HDB location derived from processes defined by the discovery service or from config file;
- Automatic re-connection and resubscription to tickerplant;
- List of tables to subscribe to supplied as configuration setting;
- More pre-built flexibility in end-of-day;
- More verbose end-of-day logging;
- Reload multiple authenticated HDBs after end-of-day;
- End-of-day save down manipulation code is shared between RDB, WDB and tickerplant log replay

See the top of the file for more information.

8.4 Write Database (WDB)

The Write Database or WDB is based on w.q[2]. This process features a number of modifications and enhancements over w.q:

1. Greater configuration options; max rows on a per table basis, list subscription tables, upd function etc. See the top of the process file for the options;
2. Use of common code with the RDB and Tickerplant Log Replay process to manipulate tables before save, sort and apply attributes;
3. Checks whether to persist data to disk on a timer rather than on each tick;
4. Informs other RDB, HDB and GW processes that end of day save and sort has completed;

[2] http://code.kx.com/wiki/Cookbook/w.q

5. More log information supplied.

The WDB process can broken down into two main functions:

1. Periodically saving data to disk and
2. Sorting data at end of day

The WDB process provides flexibility so it can be set-up as a stand-alone process that will both save and sort data or two separate processes (one that saves the data and another that will sort the data on disk). This allows greater flexibility around the end of day event as sorting data can be time consuming. It is also helps when implementing seemless rollovers (i.e. no outage window at end-of-day).

The behaviour of the WDB process is controlled by the `.wdb.mode` parameter. This should be set to one of following three values:

- saveandsort - the process will subscribe for data, periodically write data to disk and at EOD it will flush remaining data to disk before sorting it and informing GWs, RDBs and HDBs etc.
- save - the process will subscribe for data, periodically write data to disk and at EOD it will flush remaining data to disk. It will then inform its respective sort mode process to sort the data
- sort - the process will wait to get a trigger from its respective save mode process. When this is triggered it will sort the data on disk, apply attributes and the trigger a reload on the RDB, HDB and GW processes

When running a system with separate save and sort process, the sort process should be configured in the processes.csv file with a proctype of sort. The save process will check for processes with a proctype of sort when it attempts to trigger the end of day sort of the data.

8.5 Tickerplant Log Replay

The Tickerplant Log Replay script is for replaying tickerplant logs. This is useful for:

1. handling end of day save down failures;
2. handling large volumes of data (larger than can fit into RAM).

The process takes as the main input either an individual log file to replay, or a directory containing a set of log files. Amongst other functionality, the process can:

- replay specific message ranges;
- replay in manageable message chunks;

- ignore specific tables;
- modify the tables before or after they are saved;
- apply sorting and parting after all the data is written out.

The process must have some variables set (the tickerplant log file or directory, the schema file, and the on-disk database directory to write to) or it will fail on startup. These can either be set in the config file, or overridden from the command line in the usual way. An example start line would be:

```
q torq.q -debug -load code/processes/tickerlogreplay.q -p 9990 -.replay.tplogfile ../
    test/tplogs/marketdata2013.12.17 -.replay.schemafile ../test/marketdata.q -.
    replay.hdbdir ../test/hdb1
```

The tickerplant log replay script has extended usage information which can be accessed with -.replay.usage.

```
q torq.q -debug -load code/processes/tickerlogreplay.q -p 9990 -.replay.usage
```

8.6 Housekeeping

The housekeeping process is used to undertake periodic system housekeeping and maintenance, such as compressing or removing files which are no longer required. The process will run the housekeeping jobs periodically on a timer. Amongst other functionality the process:

- Allows for removing and zipping of directory files;
- Provides an inbuilt search utility and selectively searches using a 'find' and 'exclude' string, and an 'older than' parameter;
- Reads all tasks from a single CSV;
- Runs on a user defined timer;
- Can be run immediately from command line or within the process;
- Can be easily extended to include new user defined housekeeping tasks.

The process has two main parameters that should be set prior to use; runtimes and inputcsv.'Runtimes' sets the timer to run housekeeping at the set time(s), and 'Inputcsv' provides the location of the housekeeping csv file. These can either be set in the config file, or overridden via the command line. If these are not set, then default parameters are used; 12.00 and 'KDBCONFIG/housekeeping.csv' respectively. The process is designed to run from a single csv file with five headings:

- Function details the action that you wish to be carried out on the files, initially, this can be rm (remove) and zip (zipping);

- Path specifies the directory that the files are in;
- Match provides the search string to the find function, files returned will have names that match this string;
- Exclude provides a second string to the find function, and these files are excluded from the match list;
- Age is the 'older than' parameter, and the function will only be carried out on files older than the age given (in days).

An example csv file would be:

```
function,path,match,exclude,age
zip,./logs/,*.log,*tick*,2
rm,./logs/,*.log*,*tick*,4
zip,./logs/,*tick*,,1
rm,./logs/,*tick*,,3

function path      match    exclude  age
-----------------------------------------
zip      "./logs/" "*.log"  "*tick*" 2
rm       "./logs/" "*.log*" "*tick*" 4
zip      "./logs/" "*tick*" ""       1
rm       "./logs/" "*tick*" ""       3
```

The process reads in the csv file, and passes it line by line to a 'find' function; providing a dictionary of values that can be used to locate the files required. The find function takes advantage of system commands to search for the files according to the specifications in the dictionary. A search is performed for both the match string and the exclude string, and cross referenced to produce a list of files that match the parameters given. The files are then each passed to a further set of system commands to perform the task of either zipping or removing. Note that an incomplete csv or non-existant path will throw an error.

The remove and zipping functions form only basic implimentations of the housekeeping process; it is designed to be exended to include more actions than those provided. Any user function defined in the housekeeping code can be employed in the same fashion by providing the name of the function,search string and age of files to the csv.

As well as being scheduled on a timer, the process can also be run immediately. Adding '-hk.runnow 1' to the command line when starting the process will force immediate running of the actions in the housekeeping csv. Likewise, setting runnow to 1b in the config file will immediately run the cleaning process. Both methods will cause the process to exit upon completion. Calling hkrun[] from within the q process will also run the csv instructions immediately. This will not affect any timer scheduling and the process will remain open upon completion.

Housekeeping works both on windows and unix based systems. Since the process utilizes inbuilt system commands to perform maintenances, a unix/windows switch detects the operating system of the host and applies either unix or widows functions appropriately.

Extensions need only be made in the namespace of the hosting operating system (i.e. if you are using a unix system, and wish to add a new function, you do not need to add the function to the windows namespace to). Usage information can be accessed using the '-hkusage' flag:

```
q torq.q -load code/processes/housekeeping.q -p 9999 -proctype housekeeping -procname
     hk1 -debug -hkusage
```

8.7 File Alerter

The file alerter process is a long-running process which periodically scans a set of directories for user-specified files. If a matching file is found it will then carry out a user-defined function on it. The files to search for and the functions to run are read in from a csv file. Additionally, the file alerter process can:

- run more than one function on the specified file.
- optionally move the file to a new directory after running the function.
- store a table of files that have already been processed.
- run the function only on new files or run it every time the file is modified.
- ignore any matching files already on the system when the process starts and only run a function if a new file is added or a file is modified.

The file alerter process has four parameters which should be set prior to use. These parameters can either be set in the config file or overridden on the command-line. If they are not set, the default parameters will be used. The parameters are as follows.

inputcsv - The name and location of the csv file which defines the behaviour of the process. The default is /KDBCONFIG/filealerter.csv.

polltime - How often the process will scan for matching files. The default is 0D:00:01, i.e., every minute.

alreadyprocessed - The name and location of the already-processed table. The default is /KDBCONFIG/filealerterprocessed. This table will be created automatically the first time the process is ran.

skipallonstart - If this is set to `1`, it will ignore all files already on the system; if it is set to `0`, it will not. The default value is `0`.

The files to find and the functions to run are read in from a csv file created by the user. This file has five columns, which are detailed below.

path - This is the path to the directory that will be scanned for the file.

match - This is a search string matching the name of the file to be found. Wildcards can be used in this search, for example, "file*" will find all files starting with "fil".

function - This is the name of the function to be run on the file. This function must be defined in the script KDBCODE/processes/filealerter.q. If the function is not defined or fails to run, the process will throw an error and ignore that file from then on.

newonly - This is a boolean value. If it is set to `1`, it will only run the function on the file if it has been newly created. If it is set to `0`, then it will run the function every time the file is modified.

movetodirectory - This is the path of the directory you would like to move the file to after it has been processed. If this value is left blank, the file will not be moved.

It is possible to run two separate functions on the same file by adding them as separate lines in the csv file. If the file is to be moved after it is processed, the file alerter will run both functions on the file and then attempt to move it. A typical csv file to configure the file alerter would look like:

```
path,match,function,newonly,movetodirectory
/path/to/dirA,fileA.*,copy,0,/path/to/newDir
/path/to/dirB,fileB.txt,email,1,
/path/to/dirA,fileA.*,delete,0,/path/to/newDir

path            match      function  newonly    movetodirectory
-------------------------------------------------
"/path/to/dirA" "fileA.*"       copy    0       "/path/to/newDir"
"/path/to/dirB" "fileB.txt"     email   1       ""
"/path/to/dirA" "fileA.*"       delete  0       "/path/to/newDir"
```

The file alerter process reads in each line of the csv file and searches files matching the search string specified in that line. Note that there may be more than one file found if a wildcard is used in the search string. If it finds any files, it will check that they are not in the already processed table. If newonly is set to `1`, it only checks if the filename is already in the table. If newonly is set to `0`, it checks against the filename, filesize and a md5 hash of the file. The md5 hash and the filesize are used to determine if the file has been modified since it was processed last. If the found files have not been processed already, it then attempts to run the specified function to these files.

After the process has run through each line of the csv, it generates a table of all files that were processed on that run. These files are appended to the already processed table which is then saved to disk. The file alerter will attempt to move the files to the 'movetodirectory', if specified. If the file has already been moved during the process (for example, if the function to run on it was 'delete'), the file alerter will not attempt to move it.

The file alerter is designed to be extended by the user. Customised functions should be defined within the filealerter.q script. They should be diadic functions, i.e., they take two parameters: the path and the filename. As an example, a simple function to make a copy of a file in another directory could be:

```
copy:{[path;file] system "cp ", path,"/", file, " /path/to/newDir"}
```

Although the process is designed to run at regular intervals throughout the day, it can be called manually by invoking the `FArun[]` command from within the q session. Similarly, if new lines are added to the csv file, then it can be re-loaded by calling the `loadcsv[]` command from the q session.

Each stage of the process, along with any errors which may occur, are appropriately logged in the usual manner.

The file alerter process is designed to work on both Windows and Unix based systems. Since many of the functions defined will use inbuilt system command they will be need to written to suit the operating system in use. It should also be noted that Windows does not have an inbuilt md5 hashing function so the file alerter will only detect different versions of files if the filename or filesize changes.

8.8 Reporter

8.8.1 Overview

The reporter process is used to run periodic reports on specific processes. A report is the result of a query that is run on a process at a specific time. The result of the query is then handled by one of the inbuilt result handlers, with the ability to add custom result handlers.

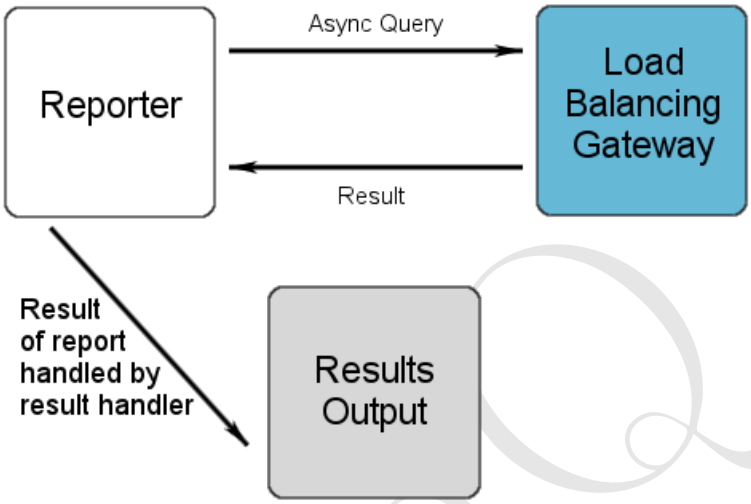

Figure 8.5: Reporter process

Features:

- Easily create a report for information that you want;
- Fully customizable scheduling such as start time, end time and days of the week;
- Run reports repeatedly with a custom period between them;
- Asynchronous querying with custom timeout intervals;
- Inbuilt result handlers allow reports to be written to file or published;
- Custom result handlers can be defined;
- Logs each step of the report process;
- Fully integrated with the TorQ gateway to allow reports to be run across backend processes.

The reporter process has three parameters that are read in on initialisation from the reporter.q file found in the $KDBCONFIG/settings directory. These settings are the string filepath of the input csv file, a boolean to output log messages and timestamp for flushing the query log table.

To run the reporter process:

```
q torq.q -load code/processes/reporter.q -p 20004
```

Once the reporter process has been initiated, the reports will be scheduled and no further input is required from the user.

8.8.2 Report Configuration

By default, the process takes its inputs from a file called reporter.csv which is found in the $KDBCONFIG directory. This allows the user complete control over the configuration of the reports. As the queries are evaluated on the target process, local variables can be referenced or foreign functions can be run. Table 8.2 shows the meaning of the csv schema.

Column header	Description and example
name	Report name e.g. Usage
query	Query to be evaluated on that process. It can be a string query or function
resulthandler	Result handlers are run on the returned result. Custom result handlers can be added. The result handler must be a monadic function with the result data being passed in e.g. writetofile["./output";"usage"]
gateway	If non null the reporter will query processes route the query to the proctype specified in this field. The values in the proctype field will be the process types on which the gateway runs the backend query. e.g. ‘gateway
joinfunction	Used to join the results when a gateway query is being used. The choice of joinfunction must take into account the result that will be received. The function must be monadic and the parameter will be the list of results returned from the backend processes e.g. raze
proctype	The type of process that the report will be run on. If the gateway field is not empty this may be a list of process types, otherwise the reporter will throw an error on startup. e.g. ‘rdb
procname	The name of a specific process to run the report on. If left null, the reporter process will select a random process with the specified proctype. If the gateway field is not null, this field specifies the specific gateway process name to run the query against e.g. ‘hdb1
start	Time on that day to start at e.g. 12:00
end	Time on that day that the report will stop at e.g. 23:00
period	The period between each report query e.g. 00:00:10
timeoutinterval	The amount of time the reporter waits before timing out a report e.g. 00:00:30
daysofweek	Numeric value required for the day of the week. Where 0 is Saturday and 2 is Monday

Table 8.2: CSV schema explanation

When running a report on a gateway, the gateway field must be set to the proctype of the gateway that will be queried. It will then run the report on the processes which are listed in the proctype field and join the results by using the function specified in the joinfunction field. If there is no join function then the reporter process will not start. Multiple entries in the proctype field must be separated by a space and are only allowed when the gateway field is not empty. If gateway field is empty and there are multiple entries in the proctype field then the reporter process will not load.

Listing 8.1 shows an example of the schema needed in the input csv file.

```
name|query|resulthandler|gateway|joinfunction|proctype|procname|start|end|period|
    timeoutinterval|daysofweek
```



```
usage|10#.usage.usage|writetofiletype["./output/";"usage";"csv"]|||rdb
    ||00:01|23:50|00:01|00:00:01|0 1 2 3 4 5 6
memory|.Q.w[]|writetofile["./output/";"memory.csv"]|||rdb|rdb1
    |00:05|18:00|00:01|00:00:08|0 1 2 3 4 5 6
usage_gateway|10#.usage.usage||gateway|raze|rdb hdb||00:02|22:00|00:01|00:00:10|0 1 2
     3 4 5 6
```

Listing 8.1: CSV example

8.8.3 Result Handlers

There are several default result handlers which are listed below. Custom result handlers can be defined as required. The result handler will be invoked with a single parameter (the result of the query).

writetofiletype - Accepts 3 parameters: path, filename, filetype and data. When writing to file it uses a date time suffix so the resultant filename will be `usage_rdb_2014_01_02_15_00_12.txt` e.g.

```
writetofiletype["./output/";"usage";"csv"]
```

splaytable - This accepts 3 parameters: path, file and data. This splays the result to a directory. The result must be a table in order to use this function e.g.

```
splaytable["./output/";"usage"]
```

emailalert - This accepts 3 parameters: period, recipient list and data. The period dictates the throttle i.e. emails will be sent at most every period. The result of the report must be a table with a single column called messages which contains the character list of the email message. This is used with the monitoring checks to raise alerts, but can be used with other functions.

```
emailalert[0D00:30;("test@aquaq.co.uk";"test1@aquaq.co.uk")]
```

emailreport - This accepts 3 parameters: temporary path, recipient list, file name, file type and data. The data is written out as the file type (e.g. csv, xml, txt, xls, json) with the given file name to the temporary path. It is then emailed to the recipient list, and the temporary file removed.

```
emailreport["./tempdir/"; ("test@aquaq.co.uk";"test1@aquaq.co.uk"); "EndOfDayReport";
     "csv"]
```

publishresult - Accepts 1 parameter and that is the data. This is discussed later in the subsection 8.8.5.

Custom result handlers can be added to $KDBCODE/processes/reporter.q . It is important to note that the result handler is referencing local functions as it is executed

in the reporter process and not the target process. When the query has been successful the result handler will be passed a dictionary with the following keys: queryid, time, name, procname, proctype and result.

8.8.4 Report Process Tracking

Each step of the query is logged by the reporter process. Each query is given a unique id and regular system messages are given the id 0. The stage column specifies what stage the query is in and these are shown in table 8.3. An appropriate log message is also shown so any problems can easily be diagnosed. The in memory table is flushed every interval depending on the value of the flushquerylogginterval variable in the reporter.q file found in the $KDBCONFIG/settings directory. An example of the query log table can be seen in listing 8.2.

Stage symbol	Explanation
R	The query is currently running
E	An error has occurred during the query
C	The query has been completed with no errors
T	The query has exceeded the timeout interval
S	System message e.g. "Reporter Process Initialised"

Table 8.3: Query Stage explaination

```
time                          | queryid stage message
------------------------------| ------------------------------------------------------------------------
2014.10.20D22:20:06.597035000| 37 R "Received result"
2014.10.20D22:20:06.600692000| 37 R "Running resulthandler"
2014.10.20D22:20:06.604455000| 37 C "Finished report"
2014.10.20D22:30:00.984572000| 38 R "Running report: rdbtablecount against proctype: rdb on handle: 7i"
2014.10.20D22:30:00.991862000| 38 R "Received result"
2014.10.20D22:30:00.995527000| 38 R "Running resulthandler"
2014.10.20D22:30:00.999236000| 38 C "Finished report"
2014.10.20D22:30:06.784419000| 39 R "Running report: rdbtablecount against proctype: rdb on handle: 7i"
2014.10.20D22:30:06.796431000| 39 R "Received result"
```

Listing 8.2: Examples of reports

8.8.5 Subscribing for Results

To publish the results of the report, the reporter process uses the pub sub functionality of TorQ. This is done by using the using the inbuilt result handler called publishresult. In order to subscribe to this feed, connect to the reporter process and send the function shown below over the handle. To subscribe to all reports use a backtick as the second parameter and to subscribe to a specific reports results include the reporter name as a symbol.


```
/- define a upd function
upd:insert

/- handle to reporter process
h: hopen 20004

/- Subscribe to all results that use the publishresult handler
h(`.ps.subscribe;`reporterprocessresults;`)

/- Subscribe to a specific report called testreport
h(`.ps.subscribe;`reporterprocessresults;`testreport)
```

8.8.6 Example reports

The following are examples of reports that could be used in the reporter process. The rdbtablecount report will run hourly and return the count of all the tables in a rdb process. The memoryusage report will run every 10 minutes against the gateway for multiple processes and will return the `.Q.w[]` information. Both of these reports run between 9:30am to 4:00pm during the weekdays. The report onetimequery is an example of a query that is run one time, in order to run a query once, the period must be the same as the difference between the start and end time.

```
name|query|resulthandler|gateway|joinfunction|proctype|procname|start|end|period|
    timeoutinterval|daysofweek
rdbtablecount|ts!count each value each ts:tables[]|{show x`result}|||rdb|rdb1
    |09:30|16:00|01:00|00:00:10|2 3 4 5 6
memoryusage|.Q.w[]|writetofile["./output/";"memory.csv"]|gateway1|{enlist raze x}|rdb
     hdb||09:30|16:00|00:10|00:00:10|2 3 4 5 6
onetimequery|10#.usage.usage|writetofile["./output/";"onetime.csv"]|||rdb
    ||10:00|10:01|00:01|00:00:10|2 3 4 5 6
```

Listing 8.3: Examples of reports

8.9 Monitor

The Monitor process is a simple process to monitor the health of the other processes in the system. It connects to each process that it finds (by default using the discovery service, though can use the static file as well) and subscribes to both heartbeats and log messages. It maintains a keyed table of heartbeats, and a table of all log messages received.

Run it with:

```
aquaq $ q torq.q -load code/processes/monitor.q -p 20001
```

It is probably advisable to run the monitor process with the -trap flag, as there may be some start up errors if the processes it is connecting to do not have the necessary heartbeating or publish/subscribe code loaded.


```
aquaq $ q torq.q -load code/processes/monitor.q -p 20001 -trap
```

The current heartbeat statuses are tracked in .hb.hb, and the log messages in logmsg

```
q)show .hb.hb
sym       procname    | time                          counter warning error
----------------------| ----------------------------------------------------
discovery discovery2  | 2014.01.07D13:24:31.848257000 893     0       0
hdb       hdb1        | 2014.01.07D13:24:31.866459000 955     0       0
rdb       rdb_europe_1| 2014.01.07D13:23:31.507203000 901     1       0
rdb       rdb1        | 2014.01.07D13:24:31.848259000 34      0       0

q)show select from logmsg where loglevel=`ERR
time                          sym  host  loglevel id      message
-------------------------------------------------------------------------------------
2014.01.07D12:25:17.457535000 hdb1 aquaq ERR      reload  "failed to reload database"
2014.01.07D13:29:28.784333000 rdb1 aquaq ERR      eodsave "failed to save tables :
    trade, quote"
```

8.9.1 HTML5 front end

A HTML5 front end has been built to display important process information that is sent from the monitor process. It uses HTML5, WebSockets and JavaScript on the front end and interacts with the monitor process in the kdb+ side. The features of the front end include:

- Heartbeat table with processes that have warnings highlighted in orange and errors in red
- Log message table displaying the last 30 errors
- Log message error chart that is by default displayed in 5 minute bins
- Chart's bin value can be changed on the fly
- Responsive design so works on all main devices i.e. phones, tablets and desktop

It is accessible by going to the url `http://HOST:PORT/.non?monitorui`
A screenshot of the front end can be seen in fig 8.6.

8.10 Compression

The compression process is a thin wrapper around the compression utility library. It allows periodic compression of whole or parts of databases (e.g. data is written out uncompressed and then compressed after a certain period of time). It uses four variables defined in KDBCONFIG/settings/compression.q which specify

- the compression configuration file to use

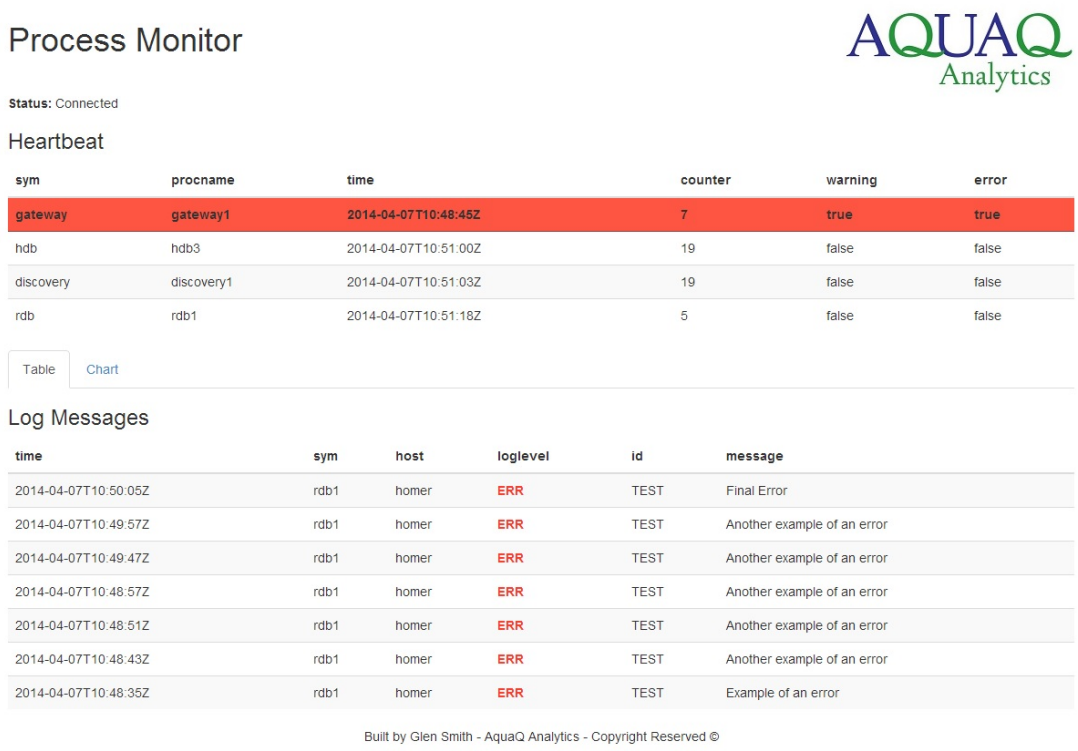

Figure 8.6: Screenshot of HTML5 front end

- the database directory to compress
- the maximum age of data to attempt to compress
- whether the process should exit upon completion

The process is run like other TorQ processes:

```
q torq.q -load code/processes/compression.q -p 20005
```

Modify the settings file or override variables from the command line as appropriate.

8.11 Kill

The kill process is used to connect to and terminate currently running processes. It kills the process by sending the exit command therefore the kill process must have appropriate permissions to send the command, and it must be able to create a connection (i.e. it will not be able to kill a blocked process in the same way that the unix command kill -9 would). By default, the kill process will connect to the discovery service(s), and

kill the processes of the specified types. The kill process can be modified to not use the discovery service and instead use the process.csv file via the configuration in the standard way.

If run without any command line parameters, kill.q will try to kill each process it finds with type defined by its .servers.CONNECTIONS variable.

```
q torq.q -load code/processes/kill.q -p 20000
```

.servers.CONNECTIONS can optionally be overridden from the command line (as can any other process variable):

```
q torq.q -load code/processes/kill.q -p 20000 -.servers.CONNECTIONS rdb tickerplant
```

The kill process can also be used to kill only specific named processes within the process types:

```
q torq.q -load code/processes/kill.q -p 20000 -killnames hdb1 hdb2
```


Chapter 9

Integration with kdb+tick

AquaQ TorQ can be fully integrated with kdb+tick. For further details, use one of the AquaQ TorQ Starter packs to set up a production kdb+ data capture system.

Chapter 10

What Can We Do For You?

AquaQ are a leading provider of kdb+ support, training and consultancy. Our staff have many years of experience architecting and implementing kdb+ systems. We would be happy to engage with you either implementing and customizing AquaQ TorQ, or in bespoke development and support of incumbent systems. Areas that we can assist include:

- Schema Design: deciding the best schema to capture, store and analyse your data;
- Real Time Data Processing: process and act on live data as fast as possible;
- Gateway Design: transparent access across heterogenous data sources e.g. real-time databases and historic database. Load balancing across homogeneous resources;
- Resilience: no single points of failure. Disaster recovery strategies;
- Massive Data Management: strategies to minimise the system memory footprint whilst maintaining access to the data;
- System Stabilisation: ensuring system stability, resolving system bottlenecks.
- Quantitative Analysis: helping you get the most from your data.

Our experience to date is predominantly in the Capital Markets industry. However, our expertise in system architecture and data analysis techniques will extend across domains into other sectors. Please contact us to talk to one of our experts.

10.1 Feedback

Please submit suggestions, improvements and bug reports to

info@aquaq.co.uk

