1. System Overview

The Quark DB Infrastructure system consists of several basic modules, including the Command Line Processor (CLP), Quark Client, Quark Server, and the Quark Query Engine. The relationship between these modules can be roughly captured by the following graph:

![Graph showing relations between CLP, Client, Server, and Query](graph.png)

Figure 1: Quark System Overview

From the viewpoint of functionality, their relations could be roughly captured in the following graph:

Figure 1 illustrates relations between different parts. Basically CLP and Client provide two different interfaces for system users to input queries. Then internally CLP and Server need to connect with a Command Processor to process them.

The Server consists of a listening thread, a thread pool, and a request queue. The listening thread is always listening on a publicly known service port and inserts a new request into the request queue such as a new client request service. The thread pool contains a fixed number of threads, and each thread retrieves the first request from the working queue and services its request.

The Client end is a lightweight design. A Quark client simply presents a GUI to users, allowing inputting commands and reviewing results. It sends the command to the server and displays the server's response on the GUI.

2. Quark Server

In this part, we will discuss how Quark implements the client-server architecture and how it supports concurrent execution from multiple clients.

2.1 Overview

As in the normal client-server scenarios, the user would communicate with the server through a client application. Since there are likely to be a lot of concurrent client requests, the server should be able to run multiple threads at the same time, each of which services a different client request.

There are various ways to implement a multithreaded server, such as thread per request and the thread pool. Thread per request means the server creates a new thread each time a
request arrives and services the request in the new thread. One of the disadvantages of the thread-per-request approach is that it involves a lot of overhead for each thread created. Each thread needs to be created and destroyed, which can be time-consuming and inefficient.

In our scenario, we want to handle a large number of concurrent client requests. Since the tasks the Quark server will be handling are mostly short-lived query requests, handling each request in a new thread is not efficient. Therefore, we will use a thread pool model. In other words, Quark server runs a fixed number of worker threads that will be reused to service new requests.

Pooled threads are usually combined with a work queue. The work queue signals waiting worker threads whenever a new job arrives. Each worker thread will either process a client's command or wait on the work queue for the next. When processing a user's command, the worker thread is performing the same operation as that shown in Figure 3.
Once a new request arrives, the listening thread will append it to the command queue and notify the waiting threads. Then one of the waiting threads will process it right away.

4.2 Implementation details

Now we turn to some implementation details with the server. In particular, we describe classes belonging to the server end and present various programming APIs provided by the server. The server is written in C++.

4.2.1 Overview of objects

The basic objects that implement the server facility include Server, Configurator, and ThreadPool. Server provides two APIs, Server::start() and Server::stop(); Server::start() initiates the server and does not return until a stop command is issued. When the stop command arrives, Server::stop() will be invoked, which will then join all threads and shut down the server. Server needs to work with a thread pool. The object ThreadPool serves this purpose. ThreadPool represents a fixed number of worker threads. Each thread in the pool in turn is associated with a ThreadHandler object.

The ThreadHandler object describes what will occur when a thread upon receiving a request. It defines a main() function to handle the request. In essence, the main() function works in a similar way to CLP. In addition, it needs to take additional care of the communication between the server and client.

Figure 2 Thread-pool model

See Figure 4 for details.
Now we will use details associated with these objects.

4.2.2 Server

The configuration of the server is done in the constructor, as shown in Figure 3. The server first reads configuration parameters from the file with the help of Configurator. Then it constructs the worker queue, the thread pool and the service socket based on the corresponding parameters. The worker queue is a regular queue plus necessary synchronization mechanisms. The service socket is the standard server socket.

ThreadPool will be discussed in details shortly.

server::start() simply loops forever waiting for connections. Each time a new request arrives, it will be inserted to the queue. Free worker threads will pick the first request from the queue and service it. Synchronization is necessary in this case to ensure correctness.

4.2.3 ThreadPool

Let’s now turn to threads part. ThreadPool defines a fixed number of worker threads. These threads are associated with a threadMain() function defined in the object ThreadHandler.

Threads in the pool are started in the constructor of ThreadPool. In other words, when Server is constructed, all the threads will be started and proceeds with executing the function ThreadHandler::threadMain(), as shown in Figure 4.

ThreadMain() has a parameter requestQueue, which corresponds to the request queue in ThreadPool::threadMain(), which corresponds to the request queue in the server::start(). The threadMain() function will loop forever on each iteration, it tries to get the first element in the worker queue and service it.
Similarly to what has been done in CommandLineProcessor, a QuarkCommandProcessor is defined in the function. Then it is passed as a parameter to the constructor of another object, ProtocolHandler. ProtocolHandler essentially controls the communication between the server and the current thread. The only function presented by ProtocolHandler is ProtocolHandler::processClient(). It takes in the client socket as the argument, and takes charge of interacting with it.

For more details about the communication protocol between clients and the server, please refer to the next section.

So far we have seen a few objects within the server component; the following figure may summarize connections among them.

![Diagram showing connections among server components](image-url)
4.2.4 Client-Server protocol

This section describes the protocol for communications between the Quark server and clients. The protocol boils down to Figure 6 depicts the protocol diagram. We will explain this protocol in detail in the following subsections.

Let us first define conventions for purposes of this document.

### 4.2.4.1 Conventions

- `\` is a space (ASCII 0x20)
- `<\cr>` is a carriage return character (0x0d) followed by a linefeed (0x0a)
- `\{A|B|C\}` exactly one of A, B, or C
- `\{foo\}` "foo" is optional

### 4.2.4.2 Escape character

Since `<\cr>` is used as an end-of-message token in both directions, we need to escape any `<\cr>` tokens that occur within actual messages. To this end, the following substitutions shall be applied to all outgoing messages (and their reverse in any received messages):

```
    \   \n    \cr   \n    \lf   \n```

### 4.2.4.3 Handshake

After accepting a connection, the server reports its protocol version to the client:

```
S: Quark{MAJOR}.{MINOR}{VENDOR_INFO}<\cr>
```

Where

- `MAJOR` is the major version number of the protocol supported
- `MINOR` is the minor version number of the protocol supported
- `VENDOR_INFO` is the vendor, or other text describing the implementation

The client then requests a particular protocol version to be used (which has to be of the same quality as the server):

```
C: VERSION{MAJOR}.{MINOR}[{FLAGS…}]<\cr>
```

Where

- `MAJOR` is the major version number of the desired protocol
- `MINOR` is the minor version number of the desired protocol
- `FLAGS` are zero or more "flags" specifying extensions to the protocol.

There are currently no valid flags defined.

This completes the protocol.
Server reports version

Client sends VERSION command

Version OK?

S: VERSION_ERROR

Client sends LOGIN command

Login OK?

Server ready to receive commands

Client sends EXEC command

Server sends results to client

S: VERSION_OK

S: LOGIN_OK

Client issues DISCONNECT command

Server sends GOODBYE message

Connection terminated

S: LOGIN_BAD_SCHEME or S: LOGIN_INVALID
4.2.4.4 Login

After the handshake is complete, the client must authenticate itself:

\[ C: \text{LOGIN} \{ \text{USERNAME} \} \{ \text{SCHEME} \} \{ \text{PARAMS} \} \{ \text{CRLF} \} \]

Where

- \text{USERNAME} is a case-sensitive username  \[ \{ \text{a-zA-Z0-9}_-+ \} \]
- \text{SCHEME} is the desired authentication scheme
- \text{PARAMS} are any parameters required by the scheme

Currently, the only supported scheme is PLAIN TEXT, and \text{PARAMS} is simply the plaintext password for the login, if any. The password has the same restrictions as the username. The space before \{ \text{PARAMS} \} is optional if there is no password.

By convention, an anonymous guest account uses the login name \text{guest} with no password.

The server responds:

\[ S: \{ \text{LOGIN_OK} | \text{LOGIN_BAD_SCHEME} | \text{LOGIN_INVALID} \} \{ \text{CRLF} \} \]

If the response is anything other than OK, the client may try authenticating again via the same scheme or another.

4.2.4.5 Executing queries or other commands

After the client has authenticated itself, it may make queries against the database server, as follows:

\[ C: \text{EXEC} \{ \text{OPTIONS} \ldots \} \{ \text{EXPR} \} \{ \text{CRLF} \} \]

Where

- \text{EXPR} is the XQuery expression to be evaluated (or other command, e.g. CREATE VIEW)

The server responds with a valid XML document, terminated with \text{CRLF}. The root of the document is either \text{<result>} (on success) or \text{<error>} (if an error occurred). The client can then execute additional queries, or disconnect.
4.2.4 Disconnecting
Although the client can simply terminate the TCP connection at any point, the “correct”
way to disconnect is by sending the DISCONNECT message:

C: DISCONNECT<CRLF>

This message can be issued at any point when the server is waiting for a client command
but not when the client is waiting for a command from the server. The server may also
terminate the connection (for example, if the client is idle for a
reason, or if the server makes excessive connections); in this case,
it should first send a GOODBYE message to the client:

S: GOODBYE<SP>{REASON}<CRLF>

Where:

REASON is an optional explanation for why the server was disconnected

4.3 User Resource Files
User resource files such as XML files that need to be queried should be placed in the
directory under the bin directory on Linux, and under the platforms/win32 directory on
Windows and Cygwin. These files can then be referred in XQuery queries in the Java or
C++ client using fn:doc(filename).