Quark System Architecture (Version 2)

Lin Guo     Feng Shao    Tony Novak
Computer Science Department
Cornell University

Copyright (C) 2003-2005 Cornell University. All Rights Reserved.
Contents

1. Introduction................................................................................................................ .3
2. System overview......................................................................................................... 3
3. Query Engine (QE)................................................................................................. 4
4. Quark Command Line Processor (CLP).............................................................. 6
5. Quark Server.......................................................................................................... 7
6. Summary................................................................................................................... 8
1. Introduction

This document is to provide an overview of the Quark system architecture. Different components of Quark will be discussed and their connections will be explained. First we will draw a picture of the entire system and then every sub-system will be described in detail.

2. System overview

The whole Quark system consists of several basic modules, including Command Line Processor (CLP), Quark Client, Quark Server and Quark Query Engine (QE). From the viewpoint of functionality, their relations could be roughly captured in the following graph:

![Figure 1 Quark System Overview](image)

Figure 1 illustrates relations between different parts. Basically CLP and Client provides two different interfaces for system users to input queries. Then internally CLP and Server need to connect to QE to process them. Now we will explain why they are connected in such a way. We start the explanation bottom up.

In the simplest terms, QE takes in a Quark command and outputs the result. In order to efficiently process a Quark command, inside QE there are other several components cooperating with each other, including Parser, Rewriter, etc. We will elaborate on this issue in section 4. Now readers could just think of QE as a black box which evaluates arbitrary commands.

CLP implements an interface which allows users to type in Quark commands through a terminal and view the result on the same screen. It simply passes user’s commands to QE and displays QE’s responses to the user.

CLP takes one command at a time and is only suitable in the single-user scenario. In order to support multi-user use of Quark, we implement a Client-Server architecture for Quark by adding Client and Server modules.

---

1. The greyed module is the storage sub-system in Quark, yet not implemented.
The Server end 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 each time a new client request arrives. The thread pool contains a fixed number of QE threads, each of which retrieves the first request from the working queue and services it.

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.

In the following section, we will examine the QE module in greater detail. And then we will discuss how CLP works with QE in section 4. Section 5 will present the overview of server design.

3. Query Engine (QE)

In this section, we will open the black box and see how a command is processed by the QE module.

The fundamental object in QE is QueryEngine. It is the interface between QE and CLP, and between QE and Server. This object presents a single function to its caller, execute(). Both the input argument and the return value of this function are strings; the input argument is the user’s command and the return value is the result of its evaluation.

Therefore the way CLP works with QE is that CLP simply passes a command to QueryEngine::execute() and reads its return value. The Server works in a similar fashion, except that there is more than one concurrent QueryEngine objects.

The object QueryEngine works as a wrapper hiding all evaluation details from users. In order to open the black box without referring to more implementation details, we expend certain functions of QueryEngine::execute(). Figure 2 shows the pseudo-code of the extended function.

You might have seen many new objects in Error! Reference source not found.. Now let us talk about them and then get back to this piece of code. The Command object represents the abstract syntax tree (AST) for a command string after being parsed by the Parser. The Query object is the YQGM representation of a Command. QueryEngine contains an EvaluationContext object (context). As its name suggests, EvaluationContext encapsulates the context needed for evaluating a command. It currently contains the following objects: Parser, Yqgm Converter, Rewriter, Optimizer and Evaluator. We will introduce those objects in turn, and then go back to the pseudo-code in Figure 2.

a) Parser parses a command string and returns a ASTCmd object. More specifically, it returns the abstract syntax tree (AST) for the command string. The main method in Parser is parse() that is declared as,

\[
\text{Command \ *} \text{parse(istream \ &\ input)};
\]

b) YqgmConverter takes in the AST of a query command and converts it to the YQGM graph. The main method with YqgmConverter is convert(), declared as:

\[
yqgm::Query \ *\text{convert(pt::Query \ *query)};
\]
Figure 2 Pseudo-code of the extended QueryEngine::execute()

```cpp
string QueryEngine::execute(string & cmdStr) {
    // parse the command
    Parser * parser = context->getParser();
    istringstream input(cmdStr);
    Command *cmd = parser->parse(input);

    // Do the YQGM conversion
    YqgmConverter *yqgmConverter = context->getYqgmConverter();
    Query *qry = yqgmConverter->convert(cmd);

    // Apply rewrite rules
    Rewriter *rewriter = context->getRewriter();
    qry = rewriter->rewrite(qry, result);

    // Apply optimizations
    optimizer::Optimizer *optimizer = context->getOptimizer();
    Query* optQuery = optimizer->optimize(qry, result);

    // Evaluate the query
    String result;
    Evaluator *evaluator = context->getEvaluator();
    evaluator->evaluate(optQuery, result);
    return result;
}
```

c) **Rewriter** takes in an YQGM graph and applies specific rewrite rules to simplify the original graph. The main method with Rewriter is rewrite(), declared as:

```cpp
yqgm::Query *rewrite(yqgm::Query * oldQry);
```

d) **Optimizer** takes in the [optionally] rewritten YQGM graph and produces an [optimized] physical evaluation plan. Then main method with Optimizer is optimize(), declared as,

```cpp
pegm::mainmemory::Query *optimize(yqgm::Query * qry);
```

e) **Evaluator** takes in a physical evaluation plan and performs the actual evaluation, returning the result as a string. The main method with Evaluator is evaluate(), declared as,

```cpp
string evaluate(pegm::mainmemory::Query * qry);
```

Let us get back to the QueryEngine::execution() now. As shown in Figure 2, given a command string, the function first gets a Parser object from its EvaluationContext. The command string is then parsed by Parser and a Command object is returned.

The following codes describe a typical query processing procedure. It consists of four phases: converting an AST to a query graph, rewriting the graph, generating the physical

---

2 This is the point from which where we extend the codes.
evaluation plan, and finally evaluating. As shown in Error! Reference source not found., first an YqgmConverter object is retrieved from the evaluation context. Then YqgmConverter performs the conversion. Next Rewriter is used to rewrite the graph. Then Optimizer is invoked to generate the physical evaluation plan. And finally, Evaluator executes this plan, and the result is returned. The entire flow chart is described in Figure 3.

![Flow Chart of Query Evaluation](image)

Figure 3. Flow Chart of Query Evaluation

Other types of commands have their different implementations of execute() according to their meanings. For more details, you should refer to the corresponding API and code.

### 4. Quark Command Line Processor (CLP)

The CLP module allows users to input Quark commands and display of results on the same terminal. After getting user’s command, it makes a function call to the QE module and passes that command as the argument. The return value of the call is the result.

The CLP module contains only one relevant class, CommandLineProcessor. This class exposes two methods. They accept commands from files and input streams, respectively. The high-level picture of how CLP works is drawn in Figure 4.

An important note about CLP is that it only contains a single QuarkCommandProcessor object and therefore it cannot be used by more than one user at the same time. We will extend this simple model in the next section.
5. Quark Server

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

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 multithread 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 the overhead of creating a new thread for each request is significant; a server that created a new thread for each request would spend more time and consume more system resources creating and destroying threads than it would processing actual user requests.

Since the tasks the Quark server will be handling are mostly short-lived query requests and there might be a large number of concurrent requests, thread-per-request is not suitable. 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 tasks.

Pooled threads are usually combined with a work queue. The work queue signals waiting threads each time a new job arrives. In our model, each “job” maps to exactly one client connection; worker threads get a socket from the queue, and service requests issued on that socket until the connection is closed. Thus, each accepted client socket will be put in the work queue and one of waiting threads will process it immediately. If the queue is already full, the listening thread will block itself and thus cannot accept new connections.

Figure 5 shows the client/thread-pool server model. The Quark server has a listening thread, a pool of worker threads, and a work queue. The listening thread is always listening on the service port. 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 in Figure 4. 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.

6. Summary
So far, we give an overview of different components of Quark system. Interested users can go on reading the more detailed documents for each component. They are stored under “doc” directory of the source directory of each component. For instance, the documents for QE are stored in quark/qe/doc/, the documents for Server are stored in quark/server/doc/, etc.