1. System Overview

The whole Quark system consists of several basic modules, including Command Line Processor (CLP), Quark Client, Quark Server, Quark Query Engine (QE), and XMLDB. These modules work together to provide a powerful query system. The relations among these modules are illustrated in the following graph:

![Quark System Overview](image)

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

2. CommandProcessor

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

2.1 Overview

The fundamental object in CommandProcessor is CommandProcessor. CommandProcessor is also the interface to the CLP and to the 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 CommandProcessor::execute() and reads its return value. The Server works in a similar fashion except that there is more than one CommandProcessor object.
2.2 Implementation details

In this section, let us further dissect the class CommandProcessor. We begin with the pseudo-code of CommandProcessor::execute():

As shown in Figure 2, given a command input stream, the CommandProcessor::execute() function first parses the command input stream and a Command object is returned. Then Command::execute() is called to perform the actual operations.

You might have seen many new objects in Figure 2. 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. Based on the different types of commands, it has several derived classes, such as:

1. QuarkQueryCmd - represents standard XQuery commands
2. QuarkInstallLibrary - represents commands to define a library module
3. QuarkSetOption/QuarkGetOption - represents commands to set or get component options
4. QuarkGetOptions - represents commands to get all available options
5. QuarkHelpCommand - represents commands that return the list of available commands

Command objects expose a single method, Command::execute(). Different derived classes have their own implementations. We will see these shortly.

Let us get back to CommandProcessor::execute(). CommandProcessor encapsulates the context needed for evaluating a command. It currently contains the following objects:

- Parser: Parses command strings and returns a Command object.
- YqgmConverter: Converts the command string to an AST.
- Rewriter: Rewrites the AST to optimize the query graph.
- Optimizer: Optimizes the query graph to generate a physical evaluation plan.
- Evaluator: Evaluates the plan and returns the result.

As mentioned earlier, CommandProcessor::execute() takes a command string as input and returns the result. Figure 2 is pseudo-code for CommandProcessor::execute().

```cpp
string CommandProcessor::execute(string cmdStr) {
    // parse the command
    Command *cmd = parser->parse(cmdStr);

    // The result string
    string result;

    // execute the command
    cmd->execute(result);

    return result;
}
```

Typically, a query processing procedure consists of four phases: converting an AST to a query graph, evaluating the graph, generating the physical evaluation plan, and finally generating the result.
evaluating. Different 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.

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:

```cpp
yqgm::Query *convert(pt::Query *query);
```

c) `Rewriter` takes in a 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 (optional) physical evaluation plan. The main method with `Optimizer` is `optimize()`, declared as:

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

e) Finally, `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);
```