1. Introduction

In Quark, a query is first parsed by the Parser to create an abstract syntax tree. Then, this syntax tree is converted to a graph from called YQGM (XQuery Graph Model) for semantic analysis. This graph is usually very complex, even for the simplest of queries, and often, due to the naïve nature of the AST-to-YQGM conversion (which, by design, operates only at the local level and thus cannot exploit patterns that emerge on a global scale) often contains irrelevant or redundant computations which would make query evaluation directly from this graph very inefficient. Therefore, we create rewrite rules which detect some of the more common patterns at several different levels of granularity, and manipulate the graph so that the query is semantically equivalent but more efficient to evaluate.

One of the important ideas behind the rewrite rules is that while many of them are very simple on their own, it is frequently the case that applying one rule “uncovers” the potential of applying another; they often interact in interesting and sometimes surprising ways. For example, the removal of an unused input column reveals the fact that its containing quantifier is not used, and removing the quantifier leaves its source operator dangling, so that the operator can be completely removed. The dangling operator removal rule does not have to capture all of this; instead, there are separate rules for removing unused columns, quantifiers, and dangling operators, and properly combined, they can make simplifications on the graph in ways that are not obviously obvious.

1.1 Rewriter rule structure

Conceptually, a rewrite rule consists of two parts:

1) **Condition** – this specifies the pattern (which may be an arbitrarily complex condition) to look for in the YQGM graph. For example, the IclOclEliminationRule, which is designed to eliminate unused output columns, has a condition function which checks if each output column is referenced somewhere; if there is at least one unreferenced output column, the condition evaluates to true.

2) **Action** – what action should be taken once the condition has been identified? For example, in the IclOclEliminationRule, once we have identified output columns that are not referenced by input columns, then these output columns can be removed from the graph. The action of each rule is defined by the function that, after it has been executed, causes the graph to be modified according to the rewrite rule.

In terms of code structure, the abstract Rule class only has one public method, `apply()`, which applies the rule to a given query graph. However, most of the rewrite rules derive from a subclass of Rule (such as `QunRule` or `OprRule`), which perform an operation over each quantifier or operator. Therefore, they must implement the following two functions:
1) **check()** – check one particular part of the YQGM graph to see if the rewrite rule applies to that part. For example, **IclOclEliminationRule::check()** takes in a Quantifier object of the kind //Quantifier as a parameter, then its parameters are unmarshaled to figure out the rewrite rule that is to be applied. Then it determines if the rewrite rule can be applied to the quantifier object. This function stores some intermediate state (to avoid recomputation) which is used by **fire()**.

2) **fire()** – this is called if **check()** returns true, and implements the rule’s action. Thus, the **apply()** method for the **QunRule** and **OprRule** simply iterates over all the quantifiers (operators) in the graph, and for each one, calls **check()** and then, if unmarshaled to result in false, does **fire()** to fire the graph unchanged.

### 2. Rewrite Rule Class Hierarchy

![Figure 1: Rewrite rule class hierarchy](image)

Figure 1 describes the class hierarchy of rewrite rules. Basically, there are two types of rewrite rules: the first is an object rewrite rule, and the second is a rule group.

#### 2.1 Single object rewrite rule types

As described before, each of the following abstract subclasses of **Rule** takes in the object it “operates” on as a parameter to its **check()** function. To implement a new rule, all we have to do is subclass one of these and implement the **check()** and **fire()** functions to reflect the rule’s conceptual condition and action:

1) **qun_rule** – a class of rules which operate on quantifiers
2) **opr_rule** – a class of rules which operate on operators

More such classes can be added if new objects develop on the YQGM graph that need to be rewritten by creating a subclass of **Rule** and implementing the **apply()** function.
2.2 Rule group types

RuleGroup is an abstract subclass of Rule, a category which encapsulates a list of other Rule objects which may be other single-object rules, or other RuleGroup objects. Activities of RuleGroup specify different semantics for how this list of rules should be applied to a graph. This may be different in comparison to the semantics of individual rules.

Currently, there are two subclasses of RuleGroup:

1) **Priority Rule Group** - There are n rules in the group, indexed 0 through n - 1.
   - For each rule i, starting with i = 0, try to apply it to the graph. If applying the rule resulted in a change in the graph, reset i to 0. The idea is that before applying each rule, we want to ensure that all rules with higher priority have been "exhausted," i.e., running them again will not have any impact.

2) **Round Robin Rule Group** - The round robin rule group applies each rule in succession until no more rules can be applied. This is similar to the Priority Rule Group except that instead of resetting the counter, the iteration is wrapped in an outer loop which terminates only when the inner loop has no effect on the graph.

In the following part, we will explain some of the individual rules that are currently used. The description of every rule consists of four parts:

1) **Purpose**
2) **What does it do? High-level description**
3) **What does it do? Low-level description**
4) **Result**: The graph property satisfied after applying the rule.

Section 3 describes the Quantifier rules; section 4 describes the Operator rules.

3. Quantifier Rules under YQGM directory

If a rewrite rule is a quantifier rule, then the rule scans every quantifier in the graph to check the applicability. Currently there is one quantifier rule, the Icl-Ocl Elimination Rule.

**Icl-Ocl Elimination Rule**

**Purpose of Rule**

The purpose of this rule is to eliminate all input columns and output columns that are not used. Since these columns are not used, they essentially will not affect the result of the query. However, it would decrease the complexity of the YQGM graph, thereby making it more efficient.

**What does it do?**

This rule scans each of the quantifiers of the graph (subclass of qun_rule). Because every input column belongs to a quantifier, we can simply remove all columns belonging to each quantifier. The result is a graph which is easier to process.

**What does it do? Low-level description**

This rule scans each of the quantifiers of the graph (subclass of qun_rule). Because every input column belongs to a quantifier, we can simply remove all columns belonging to each quantifier. The result is a graph which is easier to process.

**Result**: The graph property satisfied after applying the rule.

Section 5 describes the Quantifier rules; section 6 describes the Operator rules.

4. Operator Rules under YQGM directory

If a rewrite rule is an operator rule, it may change the order of the rules or the way they are applied.

**Literal Elimination Rule**

**Purpose of Rule**

The purpose of this rule is to remove any columns that are not used. Since these columns are not used, they essentially will not affect the result of the query. However, it would decrease the complexity of the YQGM graph, thereby making it more efficient.

**What does it do?**

This rule scans each of the quantifiers of the graph (subclass of qun_rule). Because every input column belongs to a quantifier, we can simply remove all columns belonging to each quantifier. The result is a graph which is easier to process.

**What does it do? Low-level description**

This rule scans each of the quantifiers of the graph (subclass of qun_rule). Because every input column belongs to a quantifier, we can simply remove all columns belonging to each quantifier. The result is a graph which is easier to process.

**Result**: The graph property satisfied after applying the rule.
the operator connected to the other end of the quantifier, and get all the output columns connected to that operator. If any output column is unused, we can remove it. Since every operator must have a quantifier coming out of it (otherwise that operator would be a dangling operator), this covers all the output columns. Therefore, by scanning each quantifier of the graph, we are able to scan every input column and output column.

What does it do? – Low Level Description

In Figure 2, the quantifier \( q \) connects the operator \( s_1 \) to \( s_2 \). Then for \( q \), the rule removes any input columns \( i_k \) where \( i_k \) belongs to \( q \) and \( i_k \) is not referenced by any parse tree \( p_1 \), \( p_2 \), … , \( p_n \).

Then, find all output columns \( o_1 \), \( o_2 \), … , \( o_n \) of operator \( s_1 \). Then remove any output columns \( o_k \) where \( o_k \) is not used by any input column.

Result

After the application of this rule, there should be no output columns that are not used by at least one input column. There should also be no input columns that are not used by a parse tree. In effect, every input column and output column is required on the operator they are residing on for the proper execution of the query.

4. Operator Rules under YQGM directory

Different from quantifier rules, if a rewrite rule is an operator rule, it scans every operator in the graph to check the applicability. Currently this category has five rules:

4.1 Dangling operator removal rule

Purpose of Rule

Different from quantifier rules, if a rewrite rule is an operator rule, it scans every operator in the graph to check the applicability. Currently this category has five rules:

4.1 Dangling operator removal rule
The purpose of this rule is to remove dangling operators from the graph. An operator is a dangling operator if it is not the top operator and it has no output quantifiers. Since it does not have output quantifiers, the query evaluation doesn't see any of the tuples coming out of it, and thus it does not affect the result.

**What does it do?**

**High Level description**

The rule scans each of the operators in the graph. If an operator is not the top operator and does not have any output quantifiers coming, the entire subtree rooted at the dangling operator will be removed from the graph, unless some operator outside this subtree is using the output quantifiers of the dangling operator.

**What does it do?**

**Low Level description**

The algorithm is straightforward. In check(), the number of output quantifiers is computed. In fire(), the algorithm queues all the children of the dangling operator that do not connect to any other operator in the graph, and then removes the dangling operator from the graph. After that, the algorithm visits each of the parents of the dangling operator and calls fire() on them, removing the output quantifiers of the dangling operator and any output quantifiers coming from the operator. The final step is to call fire() on all the remaining operators, if the operator has no output quantifiers (i.e., because no other operator connected to the dangling operator). We can remove all the operators in the subtree rooted at the dangling operator.

```
bool check(QueryGraph *queryGraph) {
    for (Operator *opr : queryGraph->getOperators()) {
        if (opr->isDangling()) {
            operator->removeDangling();
        }
    }
    // Call check() on each of the operators
    for (Operator *opr : queryGraph->getOperators()) {
        check(pr stunt);  // This should check if there are any dangling operators
    }
    return true;
}
```

```
bool fire(Operator *opr) {
    operator->removeDangling();
    // 1. Add any children which are not part of a common subexpression to a vector
    OprVec oprVec;
    // 2. Remove the operator itself
    operator->removeFromGraph();
    // 3. Call fire on all the children in the vector
    for (Operator *childOpr : oprVec) {
        fire(childOpr);
    }
    return true;
}
```
oprNum = oprVec.size();
for (size_t i = 0; i < oprNum; i++)
    fire(oprVec[i], memCtx, idGen, query);
return true;

Result
After the application of this rule, the operators of the graph should have at least one
input quantifier except the top one.