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 form 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 to apply a different rule; thus, a rule engine consisting of a few simple rules can be extremely powerful. As a simple example, consider the case when removing an unused input column reveals the fact that its containing quantifier is not used, and removing this quantifier leaves its source operator dangling, so that the operator can be entirely removed from the graph. 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; properly combined, they can make simplifications on the graph in ways that are not necessarily 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 pattern or a very simple one) 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 outputs columns can be removed since they aren’t used by anything and thus will not affect the result of the query. The action of each rule **must** satisfy the invariant that, after it has finished running, the query as a whole is semantically unchanged (although, naturally, this invariant may be temporarily violated while the action is running.)

In terms of code structure, the abstract Rule class only has one public method, apply(); it 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]) and therefore 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 and tests if the condition [i.e. whether or not there are any unreferenced output columns] applies to that quantifier. In this particular case, the rule stores some intermediate state (to avoid recomputation) which is later 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 [or operators] in the graph, and for each one, calls check() and fire() in succession until either check() returns false, or fire() left the graph unchanged.

All the rewriter-related code is located in src/qe/components/rewriter/.

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 single object rewrite rule, and the second is 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 needs to be rewritten, by creating a subclass of Rule and implementing the apply() function.
2.2 Rule group types

RuleGroup, an abstract subclass of Rule, is a meta-rule which encapsulates a fixed list of other Rule objects (which may be either single-object rules, or themselves rule groups); subclasses of RuleGroup specify different semantics for how this list of rules should be applied as a whole. The idea is that there are certain types of rules that can be grouped together and may interact with one another; both the grouping and the interaction are captured by the RuleGroup class. 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,” in the sense that 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.

3. GetChildElemRule – a concrete rule of Navigation Function Removal Rule\(^1\)

3.1 Review of Navigation Function Removal Rule

**Purpose of Rule**

The purpose of this rule is to remove a navigation function, e.g., getTagName() in an YQGM graph. This can be achieved if the argument(s) of the navigation function is the result of another XML construction function, e.g., createElem(). Intuitively, for getTagName(), if we find a data path, 
\[
\text{getTagName}(...(\text{createElem}(\text{elemName}, \text{elemContent}))
\]

in a query graph, we can directly use \text{elemName} as the result of getTagName without evaluating the query. And we can certainly remove the getTagName function safely.

This rewrite rule offers savings due to not constructing the intermediate XML fragments, not performing expensive navigation operations, and also creates opportunities for other

\(^1\) Also called Scalar View Composition Rule
rewrite optimizations such as Selection Pushdowns since it builds a new data path without functions.

For illustration purpose, we will show the details for removing getTagName(); other navigation functions can be removed in a very similar fashion.

**What does it do? – High Level description**

The rule consists of two phases – drill down phase and propagate phase. In the drill down phase, the rule finds out the real “result” of the navigation function by traversing the query graph. This is performed in a top-down fashion. Figure 4 shows an example of this phase for the getTagName() function.

Figure 1 shows a graph fragment that computes the tag name of a customer element. In order to find out the actual tag name, the drill down phase starts from the function getTagName($customer). It navigates down through the data references relationships and stops when it sees a createElem($name) function. At that point we know the $name is the actual result for getTagName (without evaluating the query!). Note this “drill down” path cannot contain any functions. In other words, the drill down phase stops when it sees any function, and we can only rewrite the graph if that function is a *composable* XML construction function.

After this phase, the rule then propagates up the “result” by creating pcolumns along the “drill down” path from bottom up. For the example shown in Figure 4, the rule creates an additional column for propagating up $name in each operator along the path, as shown in Figure 2.
What does it do? – Low level description

We first discuss how the two phases are implemented in the check() and fire() functions. In the check() function, the rule performs the drill down operation. It returns false if it sees a function or a constant BEFORE it reaches the real “result”, i.e., BEFORE it reaches a createElem function. Otherwise the check() function returns true.

The fire() function first does a drill down again to retrieve the result data. Note this can be optimized if we allow rules to keep states so that the check( ) function already stores the drill down path and the result column. Then the rule starts to propagate up the results. First, we create an input column in the output quantifier of the bottom operator which contains the createElem function. This new input column provides the “result” data. Next, for each operator along the “drill down path” (from bottom up), we create a pcolumn → output column → quantifier → input column to propagate the result data. Last, we stop at the operator which contains getTagName() function, and then replaces all references to the getTagName() function with ones directly referencing the “result” column.

Code structure

To make this rule extensible w.r.t. the type of operators, we implemented getSourceIcls() and propagateUpIcls() as operator methods, for drilling down and propagating up, respectively. In doing so, different types of operators can define their own rules for drilling down and propagating up columns. Essentially, they all just create an “input column → Pcolumn → output column” chain. But new operators can have very different implementation and our code structure allows it.

To make this rule extensible w.r.t the type of navigation functions, we first implemented an abstract base class which we called NavigationFunctionRemovalRule. This base rule only defines the interfaces on how to drill down and how to propagate up “result” columns. Then for different navigation functions, we have a different concrete rule (a derived class of NavigationFunctionRemovalRule). The concrete rule decides whether the “bottom” function, which is the point where drill down stops, can be “composed”. For example, getTagNameRule is a derived class of NavigationFunctionRemovalRule. In this rule, we check if the current “bottom” operator contains a createElem, and can rewrite if it’s the case. Likewise, the concrete rules also determine which parameter(s) of the bottom function is the correct result. For example, getTagNameRule will return the first argument of createElem() function while getFirstChild() will get every argument except the first one.

3.2 GetChildElemRule

This section describes another concrete rule that composes getChildElem( ) with createElem( ).

getchildElem($curElem, $childName): This function returns a sequence of child elements of $curElem with tag name
createElem($tagName, $seq1, ..., $seqN): This function creates a new element using the tag name $tagName and $seq1Æ$seqN as its contents. Note statically we cannot determine if $seqI is a child element, a text node, a namespace or other types of nodes.

The high-level idea of this rule is to use $seq1Æ$seqN to replace the function getChildElem. To do this, we first split the SPJ operator containing the createElem so that parameters $seqI can be used separately. Figure 1 shows this transformation.

Next, we just propagate up all of these columns up, until we reach the operator which contains createElem( ). Since we don’t know statically which ones of $seqI are actually child elements, so we have to look at each of the content sequences. The way to do this is setting up a correlation for each $seqi, and in the correlated subquery, we perform the following operations (bottom up):

(a) Unnest the current sequence and individual item
(b) Check if the item is an element node
(c) Check if the tag name of the element node matches the second parameter of getChildElem, which is also used in the correlated subquery
4. Concrete Unnest View Composition Rules

Purpose of Rule

The purpose of this rule is to remove an unnest operator from an YQGM graph. This can be achieved if *only* the `yqgm#unnest` function is used (note Unnest operator have other two superscalar functions `yqgm#pos` and `yqgm#last`), and if it satisfies one of the following conditions:

(a) Unnesting an XML aggregation function, e.g., `createAggSeq()`, or
(b) Uneesting a sequence construction function, e.g., `createSeq()`, or
(c) Unnesting singleton sequences.

For (a) and (b), intuitively, if we find a data path, \textit{unnest(p())} where \textit{p} is an YQGM element satisfying one of the above three conditions, we can directly propagate the input to \textit{p} and replaces all references to the original \textit{unnest} function with ones directly referencing to \textit{the input}. For (c), for each singleton sequence, Unnest produces the same one, so it can be removed.

Let’s use a simple example to illustrate (a). Assume there is an unnest operator on top of a createAggSeq function, and assume the input to the createAggSeq are three singleton sequences \((1), (2), (3)\). First, the createAggSeq() function will aggregate all of these inputs and produce a sequence of cardinality \(3—(1,2,3)\). Then the unnest function will break up this sequence, and produce three singleton sequences \(– (1), (2), \text{and} (3)\). So unnest effectively reverses the aggregate operation, and thus can be removed if we can find out and use the original input to the aggregate functions.

This rewrite rule offers savings due to not constructing the intermediate XML fragments, and also creates opportunities for other rewrite optimizations such as Selection Pushdowns since it builds a new data path without functions.

For illustration purpose, we will show the details for composing unnest with createAggSeq(); other cases can be handled in a similar fashion.

\textbf{What does it do? – High Level description}

Similar to Scalar View Composition Rule, this rule also consists of two phases – drill down phase and propagate phase. In the drill down phase, the rule finds out the real “result” of the unnest function by traversing down the query graph. This is performed in a top-down fashion. Figure 4 shows an example for this phase.
Figure 6 shows that an Unnest operator breaks up a $orders sequence, which is actually created the GroupBy operator below. The drill down phase starts from the argument of the unnest function. Similar to the drill down process in the Scalar View Composition Rule, it navigates down through the data references relationships and stops when it sees a createAggSeq($order) function. At that point we know the $order represents the actual result for unnest (without evaluating the query!). Note this “drill down” path cannot contain any functions. In other words, the drill down phase stops when it sees any function, and we can only rewrite the graph if that function is a *composable* XML functions.

After this phase, the rule will propagate up the “result”. At a high level, the propagate phase will create a parallel graph which produces the unnested version of the original graph. The parallel graph takes care of propagating up the “result”. Figure 7 shows the high level picture of creating such a parallel graph.

![Diagram](image)

**Figure 7 Creating a parallel graph**

Note the propagate phase is more complex than its counterpart in Scalar View Composition Rule. This is mainly because Unnest operator changes the cardinality of the tuples. Also note that unnest operator can propagate up other non-unnested column values. That is, only one of the output columns in an unnest operator is using a #yqgm#unnest# function to unnest an input sequence, while other columns just propagate up the input values. So when we remove the unnest operator, we need to make sure that the unnested column is still associate with the correct non-unnested values. For example, assume the original input to the unnest operator is $<(1,2), a>, <(3,4), b>$ where (1,2) and (3,4) are created in the GroupBy and unnest is unnesting them $\to$ the unnest operator will produce 4 tuples: $<1,a>, <2,a>, <3,b>, <4,b>$. In the parallel graph, since we directly
propagate up (1), (2), (3), (4) individually, we have to make sure 1 is still associated with a, not b.

The way to ensure this association is to propagate up keys along with the actual “result”, and then join back the parallel graph with its original version using the keys, to recover the associations between unnested and non-unnested columns. Figure 8 shows the final step where we join the keys of the input operator to Unnest and its unnested version in the parallel graph, to produce the correct unnested tuple.

**What does it do? – Low Level description**

As shown in Figure 7, the drill down process is very similar to the one in ScalarViewCompositionRule. In fact, it calls the same function used in Scalar View Composition Rule to find out the source of the unnest operation. So we ignore the details here.

The difference is in the propagate phase. For each operator in the path from the createAggSeq to Unnest., we need to create an unnest version of the current operator that produces the following values:

(a) the output column which produces unnested value corresponding to the original unnest column

(b) the operator corresponding to the original operator

(c) the key mapping between the original operator and the unnested one.

In terms of the code structure, each operator has a member function

applyRule(unnestedOcl : out,
          unnestedOperator: out,
which recursively calls applyRule() on its input operator, and then calls buildUnnestVersion() to propagate up the three arguments. Figure 9 shows the final graph.

**Code structure**
To make this rule extensible w.r.t. to the type of operators, we implement getSourceIcls() and buildUnnestVersion() as operator methods to find out real results and create an unnest version for itself, respectively. Different types of operators define their own rules.

To make this rule extensible w.r.t the type of unnest source parse trees, we first implemented an abstract base class which we called UnnestViewCompositionRule. This base rule only defines the interfaces on how to drill down and how to propagate up “result” columns. Then for different source functions, we have a different concrete rule (a derived class of UnnestViewCompositionRule). The concrete rule decides whether the “bottom” function, which is the point where drill down stops, can be “composed”. For example, createAggSeqRule is a derived class of UnnestViewCompositionRule.

### 4.2 createAggSeqRule
We have show most of the details for createAggSeqRule in the previous overview section. The only detail we need to bring up is, when we get the input to the group by operator, we need to create a new Unnest operator on top of it – this is to ensure that we get singleton sequences even in the case where the input to the group by is not a singleton
sequence. For example, the original input to the group by can be (1,2) and (3), and the original Unnest will thus produce (1), (2), and (3). If we just propagate up the original inputs and remove the Unnest, we will get (1, 2) and (3), which is not correct.

### 4.3 createSeqRule

This rule composes the createSeq( ) function with Unnest operator. Essentially what we have to do is to get the input parameters of createSeq and directly use them as the result of the unnest function. Again, there are some technical details to ensure the correctness. First, we need to split up the operator containing the createSeq so that we can directly reference the input parameters to createSeq(). This is similar to getChildElem Rule. Then for each of the inputs, we create an unnest to make sure we get singleton sequence. Last, we create a Union operator to union all the input. This process is shown in Figure 10.

![Diagram of createSeq composition](attachment:image.png)

**Figure 10 Composing createSeq**

### 4.4 Other rules

Other concrete rules deal with the case where the source of the unnest function is actually a singleton sequence. This case is relatively easy – we just directly use the original input, without any additional work.