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) by the AST (Abstract Syntax Tree) to YQGM conversion. The YQGM graph is usually very complex, even for the simplest of queries, and often contains unnecessary 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 to avoid these inefficiencies and materialize 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; this cycle repeats until no more rules can be applied. For example, removing an unused input column reveals the fact that its containing quantifier is not used, and removing the quantifier leaves some dangling operators, so that the operator can be entirely removed from the graph. Thus, by applying rules in an appropriate order, the YQGM graph can be simplified in ways that are not necessarily obvious.

1.1 Rewrite rule structure

Conceptually, every 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 any quantifiers, then these output columns can be removed from the graph. The result is that the graph is simplified, and the query is more efficient to evaluate. However, this process may be temporarily reduced to the action of removing.

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 on each quantifier or operator, and therefore must implement the following two functions:

- applyQuantifier
- applyOperator
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 `Quantifier` class (there are are no mentioned `quantifier` objects in the graph) 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 traverse over all the quantifiers (or operators) in the graph, and for each one, calls `check()` first, then an accessor method (i.e., `alter()` method, if any) if the graph changed.

All the mentioned code is located in `src/qe/components/rewriter/`.

2. Rewrite Rule Class Hierarchy

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()` methods to reflect the rule's conceptual condition and action.

- **qun_rule** – a class of rules which operate on quantifiers
- **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.

![Figure 1 Rewrite rule class hierarchy](image-url)
2.2 Rule group types

RuleGroup is the abstract super class of Rule. It is a metaclass which organizes a fixed list of other Rule objects which may be either single object rules, or themselves rule groups. Attributes of RuleGroup specify domain semantics for how this list of rules should be applied to a graph. This idea has been in use to some extent for a few years now in the form of ordered rewrite sets. In this version, the order of application is determined explicitly 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 in the group, 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 nothing these rules will affect my output.

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.

3. Quantifier rules

If a rewrite rule is a quantifier rule, then the rule scans every quantifier in the graph to check the applicability. Currently there are three quantifier rules:

1) Select Merge Rule

4.1. Select Merge Rule

Purpose of Rule
The purpose of this rule is to reduce the number of operators in the graph by merging two adjacent operators. This rule is applicable when they are connected by a ForEach quantifier, and the source operator does not have other output quantifiers (i.e., the source operator is not a common subexpression). The merge is carried out by removing the source operator and grafting its input quantifiers into the destination. The merge would not affect the result because the only purpose of the removed spj operator is to provide values to the second spj operator. Therefore, it acts similarly to a parse tree within the second spj operator, and can be changed into exactly that without affecting the result of the query.
What does it do? – High Level Description

The rule scans each of the quantifiers of the graph and checks if it is a ForEach quantifier connecting two SPJ operators where the source is not a common subexpression. Suppose the source is $s_1$ and the destination is $s_2$. The rule performs the following steps:

1. Graft input quantifiers of $s_1$ into $s_2$ in-place. During this process, a mapping between old input columns and new ones are created.
2. Clone the parse trees of OCLs of $s_1$ into $s_2$. Clone the predicates in $s_1$ into $s_2$.

Additionally, we need the input column mapping to ensure that all references to old input columns are properly replaced by their images.

3. Remove $s_1$.

What does it do? – Low Level Description

In Figure 2, a FOREACH quantifier $q_2$ connects SPJ $s_1$ and SPJ $s_2$. $s_1$ does not have other output quantifiers while $s_2$ could have other arbitrary input ones. The rule then performs the following actions:

1. Graft every quantifier of the source operator $s_1$ into $s_2$, in the right position. At the same time create the mapping between old input columns and new ones. So in the example in Figure 2, $s_2$’s input quantifiers would be (from left to right): $q_1$, $q_4'$, $q_5'$, $q_3$. ($q_4'$ and $q_5'$ are clones of $q_4$ and $q_5$), as shown in Figure 3. Note two new quantifiers are between $q_1$ and $q_3$.

2. For every input column $i$ of the quantifier $q_2$, and for every ECL $p$ that is referencing $i$, replace $p$ with the parse tree under $o$, where $o$ is the output column referenced by $i$.

---

![Figure 2 Select Merge Rule, before the application](image1)

![Figure 3 Select Merge Rule, after the application](image2)
Let us look at Figure 2. Suppose \( s_1 \) has an output column \( o_1 \), whose parse tree is \( p_1 \). Furthermore, \( q_2 \) has an input column \( i_1 \) referencing \( o_1 \). Then the rule will replace any expression that references \( i_1 \) with a clone of \( p_1 \).

3) If \( s_1 \) has predicates, graft them into \( s_2 \).

4) Given the mapping \( M \) created in step 1, replace any references to old input columns \( i \) (of \( q_4 \) and \( q_5 \)) with \( M(i) \).

So for example, we need to ensure \( p_1 \) now uses new input columns.

5) Remove \( s_1 \) and \( q_2 \).

Result

After the application of this rule, there should be no two \( s \) operators connected by a single \( q \) operator.

4. Operator Rules

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

1) Sandwiched SPJ operator removal rule
2) Group-By to SPJ conversion rule
3) Unnest Operator removal rule
4) Decorrelate Rule
5) Navigation Function Removal Rule
6) Unnest View Composition Rule

4.1. Redundant SPJ operator removal rule

Purpose of Rule

The purpose of this rule is to remove an SPJ operator if (a) it is sandwiched between two operators, or (b) it is the top operator in the query and just propagates data from its input.

For (a), in order to be removed, the redundant SPJ must contain \( E \) column only. This means this SPJ simply passes through tuples from the operator below to the one above, without any complex computation. For (b), the output columns of the top operator must directly connect to the output columns of its input operator position-wise. This means the top operator simply propagates the input tuples and therefore can be removed.

What does it do? – High Level Description

We first classify how to handle the case (a), where an SPJ is sandwiched.
The rule scans each of the operators in the graph. If an operator is an SPJ, and it satisfies the following conditions, then it can be removed:

1) It is indeed sandwiched. In other words, it has at least one input and at least one output quantifier.
2) Its input and output quantifiers must be ForEach.
3) It has no predicates.
4) The parse trees must all be Ecolumns.
5) It cannot be involved in correlations.

To perform the removal, the rule first clones quantifiers coming out of the sandwiched operator and have them connect the top and the bottom, thereby shortcutting the middle one. Then the old quantifiers and the sandwiched operators are removed.

It is similar to handle the case (b). The rule scans each operator in the graph. If an operator is an SPJ, and it satisfies the following conditions, then it can be removed:

1) It is the top operator in the query graph
2) It has a single input quantifier
3) Its output columns directly connect to the output columns via EColumn position-wise.

If all conditions are satisfied, the rule removes the operator and updates the query-top-operator relationship.

What does it do? – Low Level Description

In Figure 5, the SPJ operator $s_2$ is sandwiched between $s_1$ and $s_3$. Let us first take a look at the details about applicability check. In the high level description, we have enumerated five necessary conditions in order to allow the use of this rule. Among them, 1) can be done by checking if the operator has both input and output quantifiers. 2), 3) and 4) can be easily checked by the rule. The rule cannot be involved in correlations, and it cannot be involved in correlations.

In Figure 6, sandwiched SPJ removal rule after the application.
checked or not. Every related column is linked into the type of an SQL element.

Suppose we satisfy all the conditions, then the rule clones $q_1$, $q_2$, and $q_3$. Even if $q_3$ is not satisfied, the rule clones $q_1$, $q_2$, and $q_3$. However, if the conditions are not satisfied, the rule drops all the quantifiers.

Result
After the application of this rule, there should be no sandwiched SPJ operators in the graph that satisfy above conditions. The result for Figure 5 is shown in Figure 6.

4.2 Group-By to SPJ Conversion Rule

Purpose of Rule
The purpose of this rule is to replace a Group-By operator that has no grouping expression and no output columns with an SPJ operator to enable the possibility of merging the new SPJ operator using the Select-Merge Rule with other operators that use similar quantifiers.

What does it do? – High Level Description
The rule scans each operator in the graph and checks if the operator is a Group-By operator that has no grouping expression and no output columns. If the operator has no grouping expression and no output columns, the rule replaces the Group-By operator with an SPJ operator (which would have no expression tree or predicates). In the actual implementation, we replace the single Group-By operator with as many SPJ operators as there are output quantifiers from the original Group-By operator. This is because we may have multiple output quantifiers from the Group-By operator and we wish to allow for the possibility of merging each one of these SPJ operators with the corresponding operator above them.

What does it do? – Low Level Description
After checking if the rule can be applied to the Group-By operator as described above, we iterate through each output quantifier of the Group-By operator. For each output quantifier, we create a new SPJ operator, clone the output quantifier to connect from the new SPJ operator to the other operator that the original quantifier was connected to, and then remove the original quantifier from the graph. After all of the iterations, each of the output quantifiers of the Group-By operator has been removed from the graph and the Group-By operator is then removed from the graph.

4.3 Redundant Unnest Operator Removal Rule

Purpose of Rule
The purpose of the rule is to remove the UNNEST operator if it is unnesting a sequence whose cardinality is always one. Since the UNNEST operator in this case will not change the semantics, since in the result of unnesting a singleton sequence produces exactly the same sequence.

What does it do? – High Level Description
The rule scans each operator in the graph. If an operator is an unnest operator and its input sequence is always of cardinality one, and only the unnest() attribute is used, the rule will be activated. The rule will basically construct new quantifiers connecting the operators below the unnest and the one above it. Therefore the unnest operator is bypassed and then removed.

What does it do? – Low Level Description
In implementation, rather than removing the unnest operator, the rule replaces it with a new SPJ operator. The SPJ operator will have a column linking the input column the unnest operator was unnesting on to an output column referenced by the input column that initially referenced the unnest() attribute. Then newly constructed SPJ will be removed properly by the Sandwiched SPJ Removal rule.

4.4. Decorrelation rule
This transformation transforms a correlated query into an equivalent decorrelated query. A correlated query is one in which the columns referenced in the Select List of an SPJ operator are referenced in the columns of a From clause of one of its ancestor operators.

In the example shown:

**Diagram:**
- Box A is a parent of Box B (if A has an iterator i.e. a foreach quantifier over B).
- Box A is a parent of Box C (if A has a foreach quantifier over C).
- Box B is a parent of Box D.

**Terminology:**
- A box is a parent of another box if it has an iterator (i.e. a foreach quantifier) over the other box.
- A box is an ancestor of another box if it is a parent of the other box, or one of its children is an ancestor of the other box.
- A box is a descendant of another box if the other box is an ancestor of the current box.
- A box is directly correlated if it contains a correlation that references a column col1 from a table in the From clause of an ancestor. The column col1 is called the correlation column. The ancestor is the source of correlation, and the box containing the correlation (Box B in the diagram) is the destination of correlation. The correlation column can be correlated to one of its ancestors, or it is a correlated to one of its ancestors if it is correlated to the correlation column.

In the example shown:
The box labeled X is an ancestor of the one labeled Y.

Y is directly correlated to X.

Z is correlated to X (but not directly correlated).

X is the source and Y is the destination.

One tiny bit of notation:

$q = o_1 \rightarrow o_2$ is a quantifier by which operator $o_1$ provides data to
operator $o_2$; $q$ is an output quantifier of $o_1$ and an input quantifier of $o_2$.

Source: Algorithm

Overview of algorithm

A correlation can be defined as $C = (s, d, q, K, P)$, where its parameters are:

$s$: the source operator (must be an SPJ)

$d$: the destination operator (must be an SPJ)

$q$: a quantifier providing input columns to $s$; the operator providing the data is $q$.

$K$: a nonempty set of input columns belonging to $q$ which are referenced by operator $d$.

$K_{key}$: a non-empty set of input columns belonging to $q$ which compose a key.

$P$: an non-empty set of all paths in the directed graph from $d$ to $s$.

We define $K_{magic}$ to be the addition of $K$ and $K_{key}$.

Conditions for decorrelation

$C$ can be decorrelated as long all quantifiers along all paths in $P$ are FOREACH quantifiers.

The Algorithm

The algorithm then proceeds as follows:

1. Flag all operators in the graph.

2. For each path $p$:
   a. For each operator $o$ along the path $p$:
      i. If $flag = False$:
         1. $flag = True$
         2. Let $q$ be the quantifier followed into $o$ (if $o$ is a leaf, then $q$ is undefined).
         3. If $o = s$:
            1. Add a clause to $s$’s Predicate, joining each key column in $K$ with the corresponding column in $add_q$.
         4. Else:
            1. If $o = d$:
               1. Create a group by operator $opr_{magic}$.
               2. Create a quantifier $q = opr_q \rightarrow opr_{magic}$.
               3. Create a set of output columns, $O_{add}$, in $opr_{magic}$ which propagate all columns in $K$ from $opr_q$ to $opr_{magic}$.
               4. Create a set of output columns, $O_{add}$, in $o$, which pass through the columns in $O_{add}$.
               5. Add $q$ to $O_{add}$.
            2. Else, apply the abstract decorrelate() method, which operates as follows:

2. Else, apply the abstract decorrelate() method, which operates as follows:
1. If $o$ is an SPJ:

   1. Create a set of input columns $I$ coming from $q$.
   2. For each quantifier $q = o \rightarrow o_i$:
      1. Create a set of output columns $O$ coming from $I$, and add $O$ to $q$.
      2. Add $q \beta O$.
   3. If the operator has more than one input quantifier, add a Predicate $jo$ joining its input quantifiers on the key columns $K$.

2. If $o$ is a GROUP BY operator:

   1. Create a set of input columns $I$ coming from $q$.
   2. Add $I$ to the beginning (most significant) of the GroupExpression.
   3. Create a LOJ $o'$, which computes a left-outer-join of $q$ and $o$.
   4. For each quantifier $q = o \rightarrow o_i$:
      1. Change the source of $q$ to $o'$.
      2. Create a set of output columns $O$ coming from $I$, and add $O$ to $q$.
      3. Add $q \beta O$.
   5. If any of $o$'s output quantifiers $o_i$ make use of the COUNT function, then an SPJ must be placed between $o$ and $o_i$ which applies the COALESCE function to the result of COUNT(), so that NULL is replaced by 0.

3. If $o$ is an ORDER BY operator:

   1. Create a set of input columns $I$ coming from $q$.
   2. Add $I$ to the beginning (most significant) of the OrderByExpression.
   3. For each quantifier $q = o \rightarrow o_i$:
      1. Create a set of output columns $O$ coming from $I$, and add $O$ to $q$.
      2. Add $q \beta O$.

4. If $o$ is a UNION operator:

   1. Create a set of input columns $I$ coming from $q$.
   2. For each quantifier $q = o \rightarrow o_i$:
      1. Create a set of output columns $O$ coming from $I$, and add $O$ to $q$.
      2. Add $q \beta O$.

Implementation in Quark

Finding correlations

Decorrelation is implemented in Quark as a QunRule— for each quantifier $Q = o \rightarrow p$, we find correlations in the graph where $p$ is the source of a correlation. This is implemented in a way that is quite straightforward, and it could probably be made a bit more efficient. However, it basically works as follows: check() does a DFS walk of the YQGM graph, finding references to an input column of $Q$ by a descendant of $p$. If any such correlations are found, a Correlation object is added to a set, and the set is then passed to the recursive decorrelate() method for each correlation in the set.
Note: currently all operators are checked and the code is rather inefficient. Only SPJ operators which are descendants of should be checked for correlations.

Eliminating correlations—recursive algorithm

The Operator class has a method, decorrelate(), which calls decorrelate_rec(), which in turn calls a virtual method implemented by every subclass of Operator:

```cpp
bool decorrelate(Correlation *corr, QunOclvecMap &map_in, QunOclvecMap &map_out);
```

This method implements the decorrelation rules described in the above algorithm for each individual operator. (Note: as an inner class of the Operator, which has the following members:

- `source`: The source operator.
- `dest`: The destination operator.
- `correlation`: The quantifier providing the correlation columns.
- `correlation_columns`: A vector of InputColumns, these are the correlation columns.
- `query`: The query in which this correlation occurs (the root of the graph).

`map_in` is a map from quantifiers to vectors of output columns; `map_out` is analogous; as the name suggests, `map_in` is used for reading and `map_out` is used for writing new mappings.

The decorrelate() method is declared and defined in each operator's .h and .cpp file (in qe/structures/yqgm), respectively.

The decorrelated flag is set and queried using the `get_flag()` and `set_flag()` methods provided in the Operator class:

```cpp
int result = opr->set_flag(F_DECORRELATED, (void*) true);
assert(result==0);
```

```cpp
bool is_decorrelated = (bool) opr->get_flag(F_DECORRELATED);
```

(The Operator class contains a void * array of flags, which can be accessed by these two methods. All flags are initialized to NULL in the Operator constructor.)

Note: we currently traverse the tree down to the leaves; this is unnecessary. Only descendant of `d` should be decorrelated; decorrelating descendants of `d` (unless they are also correlated) is unnecessary.

4.5. Navigation Function Removal Rule

Purpose of Rule

The purpose of this rule is to remove a navigation function, e.g., `getTagName()` in an SPJ operator from 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,

```
getTagName(...(createElem(elemName, elemContent))
```

in a query graph, we can directly use `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 optimizations.

---

1 Also called Scalar View Composition Rule
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 the drill down phase for the getTagName function.

After this phase, the rule then propagates up the "result" by creating new columns along the "drill down" path from the bottom up. For the example shown in Figure 5, the rule creates an additional column for propagating up the name in each operator along the path, as shown in Figure 5.

Figure 4 Drill down of getTagName

Figure 5 Propagate phase of getTagName
We first discuss how the two phases are implemented in the check() and fire() functions. We implement them as a function to perform a drill down operation. If we reach a function or a parameter before we reach the real "result" i.e., BEFORE it reaches a certain threshold. Whenever the check() function returns true, the drill down stops.

The drill down here is a drill down in the YQGM graph. This can be done in a separate method, which we call the YQGM:drill down, and in the check() and fire() functions we have implemented. The check() function performs a drill down operation. It returns true if it finds a "result" function; otherwise it returns false.

First, we create an output column in the output quantifier of the bottom operator which contains the "result" function. We then retrieve the result data from the output quantifier of the bottom operator. We then create an input column in the output quantifier of the bottom operator which contains the "result" function and in the output quantifier of the bottom operator.

Next, for each operator along the "drill down path" (from bottom up), we create an output column in the output quantifier of the bottom operator which contains the "result" function and in the output quantifier of the bottom operator. Last, we stop at the operator which contains the getTag function, and then replace all references to the getTag function with ones directly referencing the "result" column.

Code structure

To make this rule extensible with different types of operators, we implemented the getSourceIcls() and propagateUpIcls() as operator methods, for drilling down and propagating up respectively. These methods define how to drill down and propagate up the "result" function. Essentially, they all just create an "input column -> output column -> quantifier -> input column" chain. But new operators can have very different implementations and our code structure allows it.

To make this rule extensible with different types of navigation functions, we first implemented an abstract base class which we called NavigationFunctionRemovalRule. This base class defines how to drill down and propagate up the "result" function. 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, getTagRule is a derived class of NavigationFunctionRemovalRule. In this rule, we check if the current "bottom" operator contains a getTag function, and if so, we replace all references to the getTag function with ones directly referencing the "result" column.

4.6 Unnest View Composition Rule

Purpose of Rule

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

- Unnesting an XML aggregation function, e.g. createAggSeq(), or
- Unnesting a sequence construction function, e.g. createSeq(), or
- Unnesting singleton sequences.

For (a) and (b), intuitively, if we find a data path, $\text{unnest}(\ldots p())$ where $p$ is an YQGM element satisfying one of the above three conditions, we can directly propagate the input to $p$ and replace all references to the original $\text{unnest}$ function with ones directly referencing the input.

For (c), for each singleton sequence, Unnest produces the same sequence, 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 the sequence and produce three singleton sequences. To (a) and (c), the unnest operator on top of the createAggSeq function can be removed if it can be found to affect the original input on the aggregate function.

This rewrite rule often occurs due to not containing the ancestor XML segments, and also arises opportunities for other rewrite optimizations such as Selection Pushdowns since it builds a new data path without functions.

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

**What does it do?** - High Level Description

Similar to Scalar View Composition Rule, this rule also contains two phases: drill down phase and propagate phase. In the drill down phase, the drill (bottom-up),钻下”或”Bottom-up”of the operator tree (note not recurved from the query graph). This is performed by a top-down bottom-up phase in this example for this phase.

- **Drill Down Phase**: This phase traverses the query tree from bottom to top, identifying the unnest function and its input. It determines the “result” of the unnest function by traversing down the query graph. This is performed in a top-down fashion. The rule identifies the unnest function and its input at this stage.

- **Propagate Phase**: After identifying the unnest function and its input, the drill down phase provides the necessary information for the propagate phase, which involves propagating the input to the unnest function and replacing all references to the original unnest function with ones directly referencing the input. This is performed in a bottom-up fashion, allowing the rewritten query to be constructed from the bottom up.
Figure 6 shows that an Unnest operator breaks up a $orders sequence, which is usually created by the GroupBy operator below. The drill down phase starts from the argument of the unnest function. It navigates down through the data references relationships and stops when it sees a createAggSeq($order) function. At that point, we know the $order represents the actual result without evaluating the query. Note that the "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 function.

After this phase, the rule will propagate up the "result." It will create a parallel graph which produces the unnested version of the original graph. Figure 7 shows the high level picture of creating such 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 is using a unnest function to unnest 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 associated 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. 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.

The way to ensure this association is to propagate up keys along with the actual "result" value from the parallel to the original scalar version. The original version will then be recovered by joining the parallel graph with its original version using the keys. 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.

Figure 7 Creating a parallel graph

As shown in Figure 7, the drill down process is very similar to the one in Scalar View Composition Rule. However, it calls the same function used in Scalar View Composition Rule to handle the source of the unnest operation. So we ignore the details here.
The difference is in the propagate phase. For each operator on 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 code structure, each operator has a member function

```
applyRule(unnestedOcl : out, unnestedOperator: out, keyMapping: 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.

![Figure 8: Join keys](image-url)
To make the rule extensible w.r.t. the type of operators, we implemented `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. 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`. We will show the details of how to propagate up the source of the unnest operator in the corresponding concrete subclass in `yqgm_std_rules.xml`.


Figure 9: Final graph