XATOX
(Xml Algorithm Tree tO Xquery)

Final Report

(http://users.wpi.edu/~falayo/cs561/)

CS 561 Advanced DB Systems
Francisco Alayo-Espino and Ming Jiang

i. Table of Contents

ii. Table of Figures

iii. References

Reference	**Title**	**Author(s)**
[1]	Rainbow Project Code	DSRG
[2]	http://davis.wpi.edu/dsrg/rainbow/	DSRG

1 Introduction

Currently, Rainbow is a system which focuses in the problem of mapping XML to a relational data model using generic xml query loading statements, optimizing XML order-sensitive query processing via an XML Query Algebra, and serving as a solid scalable foundation for extended XML-based applications. Rainbow optimizes XQuery processing by using a unified general-purpose XML Query Algebra named XML Algebra Tree (XAT). Rainbow Core v1.0 release includes XAT tree generation, decorrelation, optimization (rewriting), schema cleanup and execution component [2]. Our project XATOX (XAT tO Xquery) focuses on the problem of how to translate an Optimized XAT tree generated by Rainbow into an xml query. We decided to only make use of the *Opimized* XAT tree because it is the tree Rainbow users really care about.

2 Background

One of the members of our team already has been familiarized with the XAT tree of the Rainbow system from another previous project, which made the XAT system not so difficult to understand. We also meet with Ling Wang and Song Wang (members of the DSRG team) to get a better understanding of how to approach the translation from an XAT tree to an xml query statement. We downloaded the last updated Rainbow project and were able to run it without difficulties [1]. Eclipse is the programming software used by the DSRG group and by us.

3 Approach

Our work relies on the optimized XAT tree that the Rainbow system creates from xml queries. In the Optimized XAT the order of operator executions are rearranged from the original Generated XAT for a better performance. Therefore, it is not so straightforward to figure out how operators correspond to the XQuery statement, compared with Generated XAT. Instead of reading the codes and try to figure out how DSRG generate the XATs, de-correlate them and further optimized them, we read the test-cases and their corresponding optimized XAT carefully and tried to summarize the rules behind it.

All the correct test-cases can be found at:
http://davis.wpi.edu/dsrg/rainbow/RainbowCore/release.htm
Currently, we already analyzed all the test-cases in widm, Xopt and Xmark [2]. Based on these Test Cases, we define our set of rules of converting an optimized XAT tree to an XQuery. Because our rules are based on a Hash Table we create, we continue with the definition of our Hash Table.

4 XATOX's Hash Table

We use a hash table to hold the identifier of the XAT nodes, the order in which the nodes have been added to the Hash Table, which nodes already have been traversed once, the paths from the root of each XAT node, a flag which indicates to which FLWR statement the operator is assign, the name of the operator, dependency information of the nodes and the XQuery fragments which are going to be used to create the final XQuery result. The information of each XAT node is going to be created (initialized) in this table while traversing the XAT node downward the first time, and is going to be updated while traversing the XAT nodes upward. The index of the hash table is going to be the unique identifier (Rainbow's "hashCode()" of the XAT Node) of each XAT node and the columns are going to be following:

- Identifier (string): This value is the unique identifier which Rainbow assigns to each node.
- Input Order (integer): This value indicates the order in which the XAT nodes where added to our Hash Table.
- Flag_Traverse (integer): This value will indicate if the node has already been traversed (0 = default value and not traversed yet, 1 = already traversed). We need this value to traverse the entire XAT tree because of the XAT NavUnnest node which can have more than one parent node. With this value we are going to be able to visit all the nodes in an optimal orderly manner and eliminate any possible loop occurrences. From now on we refer to this value as FLAG-TRAVERSE(Operator).
- Path (String): So that we don't have to traverse the tree once again to apply our rules, we assign a path value to each node in our table. For more information about this value reference to Appendix A.

- Name (string): This is going to hold the Name of the XAT Node, to indicate which type of node it is (e.g. NavUnnest, Collection, Cartesian Product, and others specified by Rainbow). These values are going to be used while implementing the XATOX rules. From now on we refer to this value as NAME(Operator).
- Flag_FLWR (integer): This is an identifier, to differentiate nested FLWR from their outer FLWR structures (in the case that our XQuery Statement has nested FLWR statements).
- Inputs/Dependents (array): These are the inputs of the XAT Nodes. A XAT Node will depend of these inputs in order to create its XQuery fragment. From now on we refer to the array of dependants of a node with Name Operator as DEPENDENTS(Operator).
- Output (string): This is the output of the XAT Nodes. Many outputs of the XAT tree are the inputs of other nodes. If a XAT Node's output is the input of another XAT Node, then the XAT Node with the output must be bellow the XAT Node with the input in the XAT Tree. From now on we refer to this value as OUTPUT(Operator).

- ❖ Pre-Value (string): This attribute is going to hold a fragment of the Rainbow's Node specification which is going to be used to create the Final-Value of the operator. Its default and initial value is blank. It is not necessary to keep this value, but we do so to make a clear demonstration of how we translate from XAT tO Xquery. From now on we refer to this value as PRE-VALUE(Operator).
- ❖ Final-Value (string): This attribute is going to hold a fragment (derived from Pre-Value) of the XQuery which is going to be used to create the complete XQuery. Its default and initial value is blank. A XATOX rule is applied to a XAT Node only if this value is empty. If the XAT Node already has this value, it implies that a rule already has been applied to the XAT Node and there is no need to apply the same rule again. From now on we refer to this value as FINAL-VALUE(Operator).

We use the Rainbow's XAT tree to traverse the operators in an orderly manner, and the hash table to know which XAT Nodes we have already have traversed, retrieving the dependency information of the XAT Nodes and store the fragments for the final XQuery result (Final-Value). After we have the necessary fragments of the final XQuery, we store them into an FLWRStructure object. Because there is the possibility of having nested FLWR, our class FLWRStructure can be nested also. As a final step, we read our FLWRStructure objects and print save it on a predefined file (XATOX.txt) and show it on Rainbow's GUI.

5 Phases

To translate XAT tO Xquery, we divided our project into three PHASEs for simplification:

1. PHASE0: Traverse Down the tree and add the XAT Node's values to our hash table.
2. PHASE1: Traverse Up the tree and apply the rules (specified in the next section) to the nodes. By applying the rules we are going to fill the FINAL-VALUE (XQuery fragments) of every important operator in our hash table and store it in a FLWR structure.
3. PHASE2: Print the XQuery result into a predefined file (XATOX.txt).

6 Rules

We already used more than half of the Test Cases available to get these rules. We recognized 23 operators, but because of time limit we were able to define the rules only for Xopt and Xmark Test Cases.

The following are simplified functions we are going to define our rules:

NAME(Operator) => Returns the NAME of Operator.
OUTPUT(Operator) => Returns the OUTPUT value of Operator.
DEPENDENTS(Operator)[0] => Returns the first dependent of Operator.
PRE-VALUE(Operator) => Returns the PRE-VALUE of Operator.
FINAL-VALUE(Operator) => Returns the FINAL-VALUE of Operator.
Nested-FLWR(Operator) => Returns Operator's corresponding nested FLWRStructure in a String format. This function is reserved only for the rules of Combine and GroupBy (with Inner Tree equal to Combine).

Because we need a substring from the PRE-VALUE which defines an XPath string (used only for the rules of NavCollection and OuterNavCollection) de additionally define the following function:

DESTINATION(Operator) => Returns the DESTINATION of Operator.

Because we need to differentiate the Inner Tree nodes from the rest, Rainbow creates them not as a common XAT node operator (this function is used only for the rule of GroupBy in our Test Cases). To retrieve information of the inner we define the following function:

InnerTree[GroupBy] => Returns the Inner Tree operator of GroupBy.

In the following list of rules for PHASE1, "Operator1 + Operator2" means that Operator2 is the descendant of Operator1 and that one of the DEPENDENTS of Operator1 corresponds to the OUTPUT of Operator2. And "Operator1 & Operator2" means that Operator 2 is in the Inner Tree of Operator1 (used for the GroupBy rule). PHASE1 will complete the XQuery fragments and locate these XQuery fragments in their corresponding locations of our generated XQuery statement (by using the FLWRStructure type). Inside each rule, we differentiate the ones that modify the FINAL-VALUEs in our Hash Table from the ones that modify the FLWR structures.

6.1 Block

- *Unfortunately we weren't able to find this operator in Rainbow's Optimized XAT tree of the test cases of Xopt and Xmark.*

6.2 Cartesian_Product

FLWR:

- ❖ Do nothing to Cartesian_Product's FINAL-VALUE because NavUnnest and GroupBy rules will take care if it is a Composed FOR or Nested FOR.

6.3 Combine

FLWR:

- ❖ Store Combine's DEPENDENTS[0]'s FINAL-VALUE into the RETURN statement. Also means that *the nested FLWR statement is ready*. If there is another operator on top of Combine (e.g. Tagger or Count), then their FINAL-VALUE should include the whole nested FLWR statement (as a string format, because it is going to be Combine's FINAL-VALUE). Every XAT node on top of the XAT Combine node is from an outer FLWR statement (therefore every node that is below the XAT Combine node is from the inner FLWR statement). We define the nested FLWR by updating the Flag_FLWR values of our hash table. Combine and GroupBy are the only operators which are going to specify if there is a *nested* FLWR statement as follows: (SPECIAL RULE FOR SELECT)

FINAL-VALUE(Combine) := Nested-FLWR(Combine)

6.4 count

VALUE:

- ❖ Retrieve count DEPENDENTS[0]'s FINAL-VALUE and store it on count's FINAL-VALUE as follows:

FINAL-VALUE(count) := **count(**FINAL-VALUE(DEPENDENTS(count)[0])**)**

6.5 Distinct

- ❖ *Unfortunately we weren't able to find this operator in Rainbow's Optimized XAT tree of the test cases of Xopt and Xmark.*

6.6 Expose

- ❖ *Unfortunately we weren't able to find this operator in Rainbow's Optimized XAT tree of the test cases of Xopt and Xmark.*

6.7 For

- *Unfortunately we weren't able to find this operator in Rainbow's Optimized XAT tree of the test cases of Xopt and Xmark.*

6.8 GroupBy

VALUE:

- GroupBy & InnerTree[GroupBy]
  - If InnerTree[GroupBy] is the function operator ***count***, then retrieve GroupBy DEPENDENTS[0]'s FINAL-VALUE to substitute on their corresponding location of GroupBy's PRE-VALUE and store it on GroupBy's FINAL-VALUE as follows:

    FINAL-VALUE(GroupBy) :=
    count(FINAL-VALUE(DEPENDENTS(GroupBy)[0])**)**

  - If InnerTree[GroupBy] is the function operator ***position***, then retrieve GroupBy DEPENDENTS[0]'s FINAL-VALUE to substitute on their corresponding location of GroupBy's PRE-VALUE and store it on GroupBy's FINAL-VALUE as follows:

    FINAL-VALUE(GroupBy) :=
    position[FINAL-VALUE(DEPENDENTS(GroupBy)[0])**]**

  - Only when InnerTree[GroupBy] is a ***Combine*** operator then one must look for the *Cartesian_Product* operator between the GroupBy operator a NavUnnest operator which has the OUTPUT equal to GroupBy's DEPENDENTS[0].
    - If found, it is a nested FLWR with the NavUnnest operator as the exclusive outer FLWR (this information is important to create the initial values of the outer FLWR). Also, every operator between the GroupBy and the NavUnnest operators is from the inner FLWR, and GroupBy's ancestor operators are from the outer FLWR. *Also means that the nested FLWR statement is ready*. GroupBy and Combine are the only operators which are going to specify if there is a *nested* FLWR statement as follows:

      FINAL-VALUE(GroupBy) := Nested-FLWR(GroupBy)

    - If not found then the FINAL-VALUE of GroupBy becomes the FINAL-VALUE of the operator that has the same OUTPUT as GrouBy as the following algorithm specifies: (SPECIAL RULE FOR SELECT)

      If OUTPUT(GroupBy) == **$col123456**
      And OUTPUT(Operator) == **$col123456**
      Then FINAL-VALUE(GroupBy) := FINAL-VALUE(Operator)

6.9 Join

VALUE:

- Join + NavCollection/OuterNavCollection
  - Retrieve NavCollection/OuterNavCollection's FINAL-VALUE to substitute on their corresponding location of the Join's PRE-VALUE as follows:

```
      If PRE-VALUE(Join) == $col123456 == $col654321
           And OUTPUT(Operator1) == $col123456
   And FINAL-VALUE(Operator1) == $person/name/text()
           And OUTPUT(Operator2) == $col654321
   And FINAL-VALUE(Operator2) == $worker/name/text()
Then FINAL-VALUE(Join) := $person/name/text() == $worker/name/text()
```

FLWR:

- Join + NavCollection/OuterNavCollection
  - Locate Join's FINAL-VALUE in WHERE statement. We should take into consideration a collection of Join operators for future implementation (e.g. one of the sons of a XAT Join operator could be another Join).

6.10 LOJ

- Ignore this operator because its only purpose is performance.

6.11 Merge

- *Unfortunately we weren't able to find this operator in Rainbow's Optimized XAT tree of the test cases of Xopt and Xmark.*

6.12 Name_Column

- *Unfortunately we weren't able to find this operator in Rainbow's Optimized XAT tree of the test cases of Xopt and Xmark.*

6.13 NatureJoin

- Ignore this operator because its only purpose is to join two edges of the XAT tree.

6.14 NavCollection/OuterNavCollection

- ❖ This rule also applies to OuterNavCollection.

VALUE:

- ❖ NavCollection/OuterNavCollection + NavUnnest
  - ➢ Do nothing, just use self DEPENDENTS[0] value as follows:

FINAL-VALUE(NavCollection) :=
DEPENDENTS(NavCollection)[0]/DESTINATION(NavCollection)

- ❖ NavCollection/OuterNavCollection NOT(+ NavUnnest)
  - ➢ It means it is a temporary value generated by Rainbow. The FINAL-VALUE for this rule would be:

FINAL-VALUE(NavCollection) :=
FINAL-VALUE(DEPENDENTS(NavCollection)[0])/DESTINATION(NavCollection)

6.15 NavUnnest

VALUE:

- ❖ For NavUnnest we use the DEPENDENTS[0]'s FINAL-VALUE to insert it in the PRE-VALUE(NavUnnest) and store it in NavUnnest's FINAL-VALUE as follows:

FINAL-VALUE(NavUnnest) := OUTPUT(NavUnnest)
IN FINAL-VALUE(DEPENDENTS(NavUnnest)[0])/DESTINATION(NavUnnest)

FLWR:

- ❖ If NavUnnest's OUTPUT is the DECENDENTS[0] of a GroupBy operator and there is a Cartesian Product between them, then store the FINAL-VALUE(NavUnnest) into the *outer* FOR fragment (it will have to be temporary in this case because the FOR fragment is going to be currently used by the inner FLWR, so after the inner FLWR statement is ready I can initiate a new outer FLWR with this initial outer FOR fragment).
- ❖ If NavUnnest's OUTPUT is not the DECENDENTS[0] of a GroupBy operator or there is not a Cartesian Product between them, store FINAL-VALUE(NavUnnest) into the *inner* FOR fragment.

6.16 OrderBy

FLWR:

- ❖ Always store FINAL-VALUE(OrderBy) into SORTBY statement.

6.17 OuterNavCollection/NavCollection

- ❖ Please reference to the rules of NavCollection.

6.18 OuterNavUnnest

- ❖ *Unfortunately we weren't able to find this operator in Rainbow's Optimized XAT tree of the test cases of Xopt and Xmark.*

6.19 position

VALUE:

- ❖ Retrieve position DEPENDENTS[0]'s FINAL-VALUE store it on position's FINAL-VALUE as follows:

  FINAL-VALUE(count) := **count(**FINAL-VALUE(DEPENDENTS(count)[0]))

6.20 Select

VALUE:

- ❖ Select + NavCollection/OuterNavCollection
  - ➢ Use this operator's FINAL-VALUE to insert on their corresponding location of the Select's PRE-VALUE and store the result as the Select's FINAL-VALUE as follows:

    If PRE-VALUE(Select) == **$col123456 == "Tom Hanks"**
    And OUTPUT(Operator) == **$col123456**
    And FINAL-VALUE(Operator) == **$person/name/text()**
    Then FINAL-VALUE(Select) == **$person/name/text() == "Tom Hanks"**

- ❖ Select + GroupBy
  - ➢ If InnerTree[GroupBy] is the function operator *position*, then:

    If PRE-VALUE(Select) == **$col614003 == 2.0**
    And PRE-VALUE(GroupBy) == **[$players]:$col614003**
    Then FINAL-VALUE(Select) == **$players[2.0]**

FLWR:

- ❖ Select + NavCollection/OutNavCollection/count
  - ➢ Store Select's FINAL-VALUE in the WHERE statement.

6.21 Source

VALUE:

- For the source, we will retrieve its PRE-VALUE and combine it with some predefined strings, because when we need to write the XQuery statement the source will always appear this way, and store it into the FINAL-VALUE as follows:

FINAL-VALUE(Source) := **document(**“PRE-VALUE(Source)”**)**

6.22 Tagger

VALUE:

- We get each DEPENDENTS’s FINAL-VALUE and put it on their corresponding location of the Tagger’s PRE-VALUE and store the result in Tagger’s FINAL-VALUE. But if a Combine’s OUTPUT equals to one of the Tagger’s DEPENDENTS then do nothing as the following algorithm shows:

```
If PRE-VALUE(Tagger) == <person>$col123456</person>
    And OUTPUT(Operator) == $col123456
    And FINAL-VALUE(Operator) == $person/name
    And Operator <> Combine
Then FINAL-VALUE(Tagger) == <person>$person/name</person>
```

FLWR:

- If a Combine’s OUTPUT equals to one of the Tagger’s DEPENDENTS then add tags to the whole FLWR statement (Combine’s FINAL-VALUE) as in PHASE 1.

6.23 XMLUnion

VALUE:

- Find all children Taggers operators and put their FINAL-VALUEs together (from right to left) and key them (concatenate “{” and “}” to the edges of the string) as follows:

FINAL-VALUE(XMLUnion) := **{**FINAL-VALUE(Tagger1)**,**FINAL-VALUE(Tagger1)**}**

7 Implementation

7.1 Sources

The following are projects used from Rainbow to implement our XATOX project:

- BasicGui: We used the ReWriteGUI class to call our classes. We made this link so that we could use the original friendly GUI of Rainbow. It served us very much while traversing the XAT Tree because we could compare our printed debug with the XAT Tree graphs given by Rainbow.
- Visual Exploration of XQuery Processing: We used the XATParser class to get an understanding of how the XAT Nodes were created. We created our own parseNode function to create the values which are going to be inserted in our XATOX table from the XAT Nodes (reference to the section that talks about our Hash Table for more detail). We used the ExecutionGUI class to add the "XATOX" Tab (specified in our new XATOX_DocViewStep class) to Rainbow's interface and removed unnecessary ones.
- RainbowCore: We used all the classes which were helpful/necessary to traverse the entire XAT Tree and get information of the XAT Nodes.

7.2 XATOX Classes

Inside the RainbowCore project we created our XATOX package. The following are classes created to implement our XATOX project:

- XATOX_Test: This is the class called by Rainbow's ReWriteGUI class and is also the class that calls XATOX_PHASE0. This class creates the Hash Table. If we don't want to use Rainbow's GUI we could execute the main function of this class with some parameters to demonstrate our project.
- XATOX_PHASE0: This class is called by XATOX_Test class and is also the class that calls XATOX_PHASE1. This class traverses the XAT Tree, states the recognizable names of the XAT Nodes and initializes the values of the Hash Table.
- Table_hash: This class is called by the XATOX_PHASE0 class. This class creates/initializes the Hash Table.
- Table_value: This class is called by the XATOX_PHASE0 class. This class creates and updates the values which are going to be inserted and updated to the Hash Table.
- XATOX_PHASE1: This class is called by XATOX_PHASE0 when we want to apply the rules of PHASE1 to a specific XAT Node. If the XAT Node already has an XQuery fragment value in our hash table, then no rule is applied to the traversed XAT Node. All the rules of PHASE1 are specified/defined in this class and the selectivity of the rule corresponds to the name of the traversed XAT Node. This class also assigns string to their corresponding FLWRStructures.
- FLWRStructure: This class is called by XATOX_PHASE2 to store the final XQuery Statement from the values in Hash Table. We represent the final XQuery Statement as a class in case there is a nested FLWR Statement.
- XATOX_PHASE2: This class is called by XATOX_PHASE0 when all the necessary XQuery fragments to create the final XQuery Statement are in our Hash Table. This will happen when there is no FLWR structure to process and the entire XQuery Statement is located in one of the rows of our Hash Table with node identification equal the operator located below the Expose XAT Node (the root of the XAT Trees).

7.3 XATOX Architecture

The following diagram depicts the XATOX environment architecture.

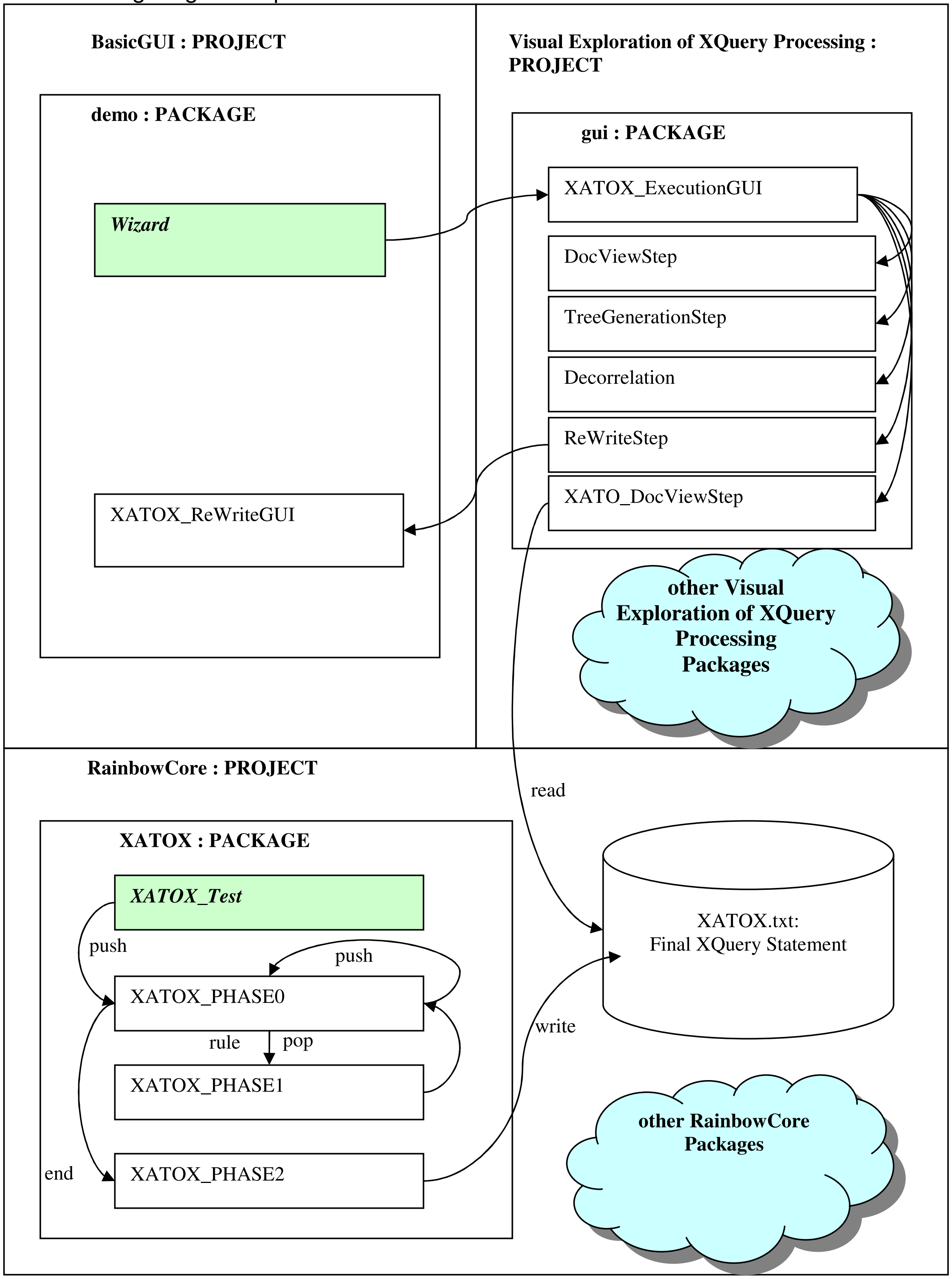


Figure 1 XATOX Environmental Architecture

The following diagram depicts the XATOX architecture:

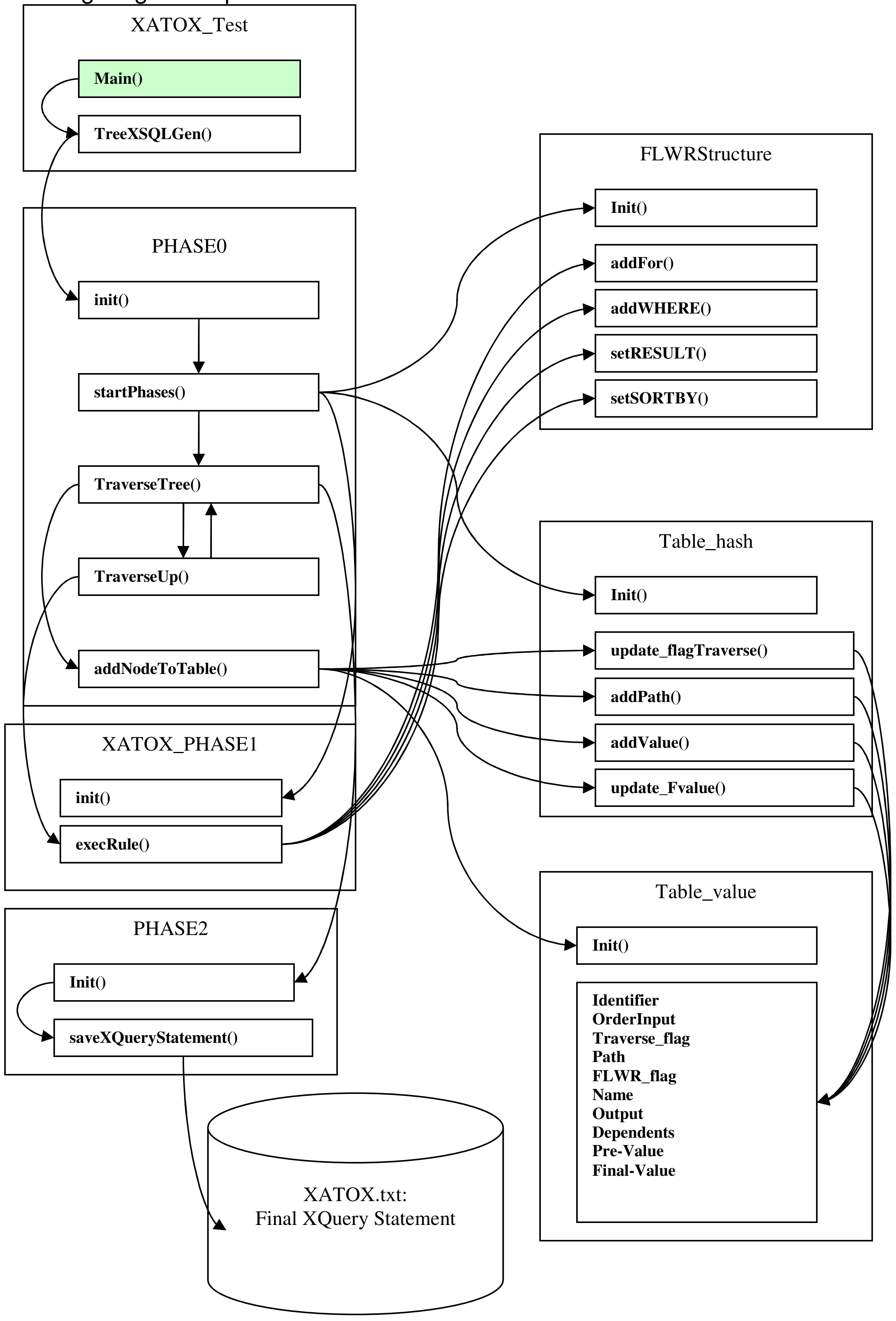


Figure 2 XATOX Architecture

7.4 Rule Algorithm Resources

For simplification of the rule algorithms, we define the following functions:

- findLink(ID0, "Operator"): It will return an array list with the ID values of the nodes which have their XAT Node names equal to "Operator" and which OUTPUT values are equal to one of the values of the DEPENDENTS of the operator with identification is ID0 (parameter). If we want to find all the DEPENDENTS without an operator' name restriction, the second parameter should be changed to "*".

- Replace(String, subString, Fragment): First we know that subString is contained in String. So we replace the subString portion of String with Fragment and return the modified String value.

- innerTree(groupByID): Returns the name of the InnerTree operator.

- destination(ID): The output of this function will be the DESTINATION(Operator) value which is used to complete the FINAL-VALUE of the operator with XAT identification ID. The function has to deal with four situations:
    a. When there is a "#text", then it should be converted into: "text()"
    b. When there is a null, do not add it.
    c. When there is "////", then it should be converted into: "//".

We can differentiate the attribute from the sub-element, by inferring to an attribute with the character "@". But not implemented yet.

8. Conclusion

Altogether, we studied the test cases of Xopt, Xmark and widm. Currently our system support Xopt except (test 6,7,12,12-2,13, all because of not available time to finish the nested FLWR rules). For the detailed discussion of nested XQuery, please refer to *Appendix C*. We think it also support some other test cases in Xmark and widm, but did not thoroughly check them.

Through this, we look deeper into operators other than just statements. While trying to find relationships between and therefore mapping these two, we learned more about XQuery and how query optimizations can occur in various occasions.

8 Appendix A: Discussion about Path value

The following is an example of path values assigned to each node in our table (Test Case 2_p of Xopt).

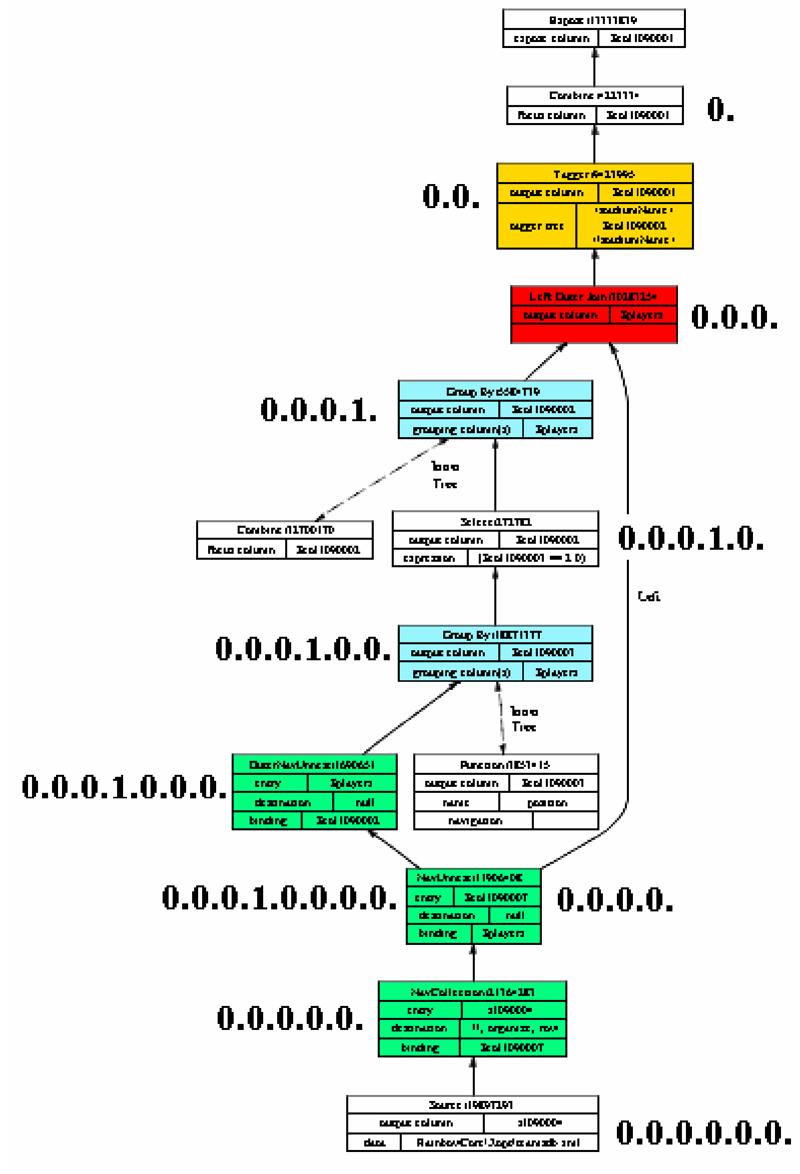


Figure 3 Path demonstration

We do not assign values to the inner tree nodes, and the only nodes which can be assigned two values are the NavUnnest XAT nodes as shown in the above figure. By assigning two values to NavUnnest we could say that we are virtually assigning 2 values to each node below it. For example if we wanted to know if the XAT node NavCollection (with 0.0.0.0.0. as assigned path) is a descendant from GroupBy (with 0.0.0.1.0.0. as assigned path), and by parsing the path values we know that NavUnnest is the ancestor of NavCollection and that NavUnnest is the descendant of GroupBy, so one can conclude that NavUnnest is the descendant of GroupBy. This path value is important to indicate if our final XQuery Statement has nested FLWR statements by applying XATOX rules at NavUnnest and GroupBy (reference to the rules section for more detail).

9 Appendix B: Differentiating inner FLWR from outer FLWR

Unfortunately we did not make it to implement nested XQuery. But based on our study of TestCases, we have found out the way to differentiate the inner FLWR from outer FLWR, which is the key to implement nested XQuery. Because if we know which operators are corresponding to the same level of XQuery and which level it is, we can apply our rules to them and get XQuery statements for each level in order. The following are some discussions and results of our observation.

The most obvious feature of nested XQuery statements is, there are always a "GroupBy"(with "Combine" inner tree) and a corresponding Cartesian Product. Let us give it some theoretical analysis first. Cartesian Product means there are two bindings, but merely by this we can not know the relationships of these two bindings: nested or same level. GroupBy with Combine always means to "combine" the results of all the descendants' XQuery fragments into a string. So when such a GroupBy and Cartesian cooperate, on one hand GroupBy requires all its descendants contribute some XQuery results and put them together, on the other hand, some of its descendants are also required by some nodes who are even ancesters of "GroupBy". Then two bindings (for there is a cartesian) on now on different level and nested XQuery appears.

However, it is quite hard to utilize this information: we can not trace the corresponding "Cartesian Product" by the value and we even can not be sure whether a GroupBy and Cartesian correspond to each other or not, nor to say we have to make sure that GroupBy have to be with "Combine". The situation tends to be more and more complicated if there are many GroupBys and Cartesian Products. In our Progress Report we realize this problem but have not soved it.

Then we can look at the tree in a different view: Each GroupBy, especially those with "Combine", have to operate on a bound variable, which is provided by a certain NavUnnest. So given a GroupBy/NavUnnest, we can use value to trace the corresponding /NavUnnest/GroupBy. Then we can try to search whether there is a Cartesian Product between them. If there is one, then there must be a nested FLWR. As we can see in figure 4, it is to make sure that there is a nested XQuery.

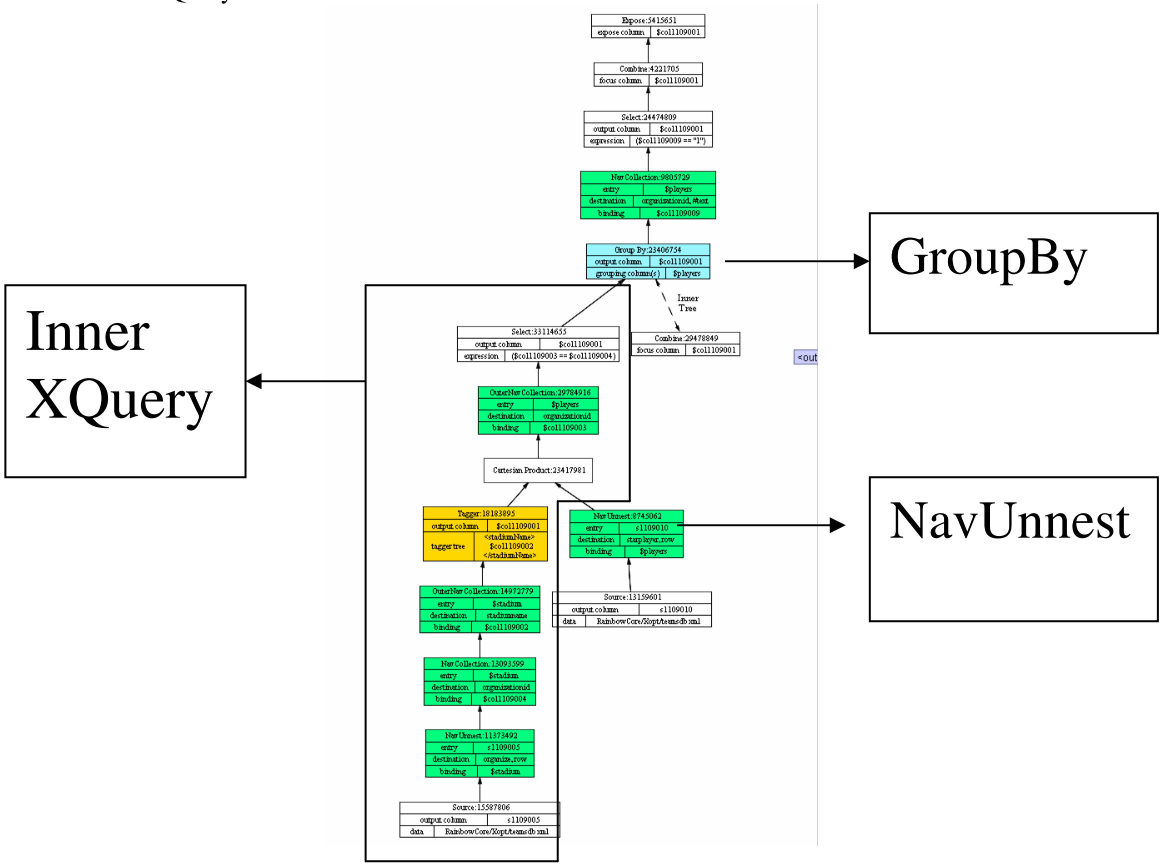

Expose:5415651
expose column | $col1109001
Combine:4221705
focus column | $col1109001
Select:24474809
output column | $col1109001
expression | ($col1109009 == "1")
Nav Collection:9805729
entry | $players
destination | organizationid, #text
binding | $col1109009
Group By:23406754
output column | $col1109001
grouping column(s) | $players
GroupBy
Inner Tree
Combine:29478849
focus column | $col1109001
<out
Inner XQuery
Select:33114655
output column | $col1109001
expression | ($col1109003 == $col1109004)
OuterNav Collection:29784916
entry | $players
destination | organizationid
binding | $col1109003
Cartesian Product:23417981
Tagger:18183895
output column | $col1109001
tagger tree | <stadiumName> $col1109002 </stadiumName>
NavUnnest:8745062
entry | s1109010
destination | starplayer,row
binding | $players
NavUnnest
Source:13159601
output column | s1109010
data | RainbowCore/Xopt/teamsdb.xml
OuterNav Collection:14972779
entry | $stadium
destination | stadiumname
binding | $col1109002
Nav Collection:13093599
entry | $stadium
destination | organizationid
binding | $col1109004
NavUnnest:11373492
entry | s1109005
destination | organize,row
binding | $stadium
Source:15587806
output column | s1109005
data | RainbowCore/Xopt/teamsdb.xml

Figure 4 Xopt 6: inner and outer

Then the second problem comes out, which operators corresponds to the inner, and which correspond to the outer? The answer is quite easy. As we can see in figure 4, since the corresponding NavUnnest is needed in the ourter XQuery, it should belong to the outer. All the operators above "GroupBy" belong to the outer. All the other operators will make some results and GroupBy will use them and feed them to "Combine" a string. So they all belong to the inner.

One may have question as: is it possible that there are some other operators between Cartesian Product and corresponding NavUnnest? If there are some, do they belong to the inner or outer? We look through the TestCases and find only one such example, which will be shown in figure 5:

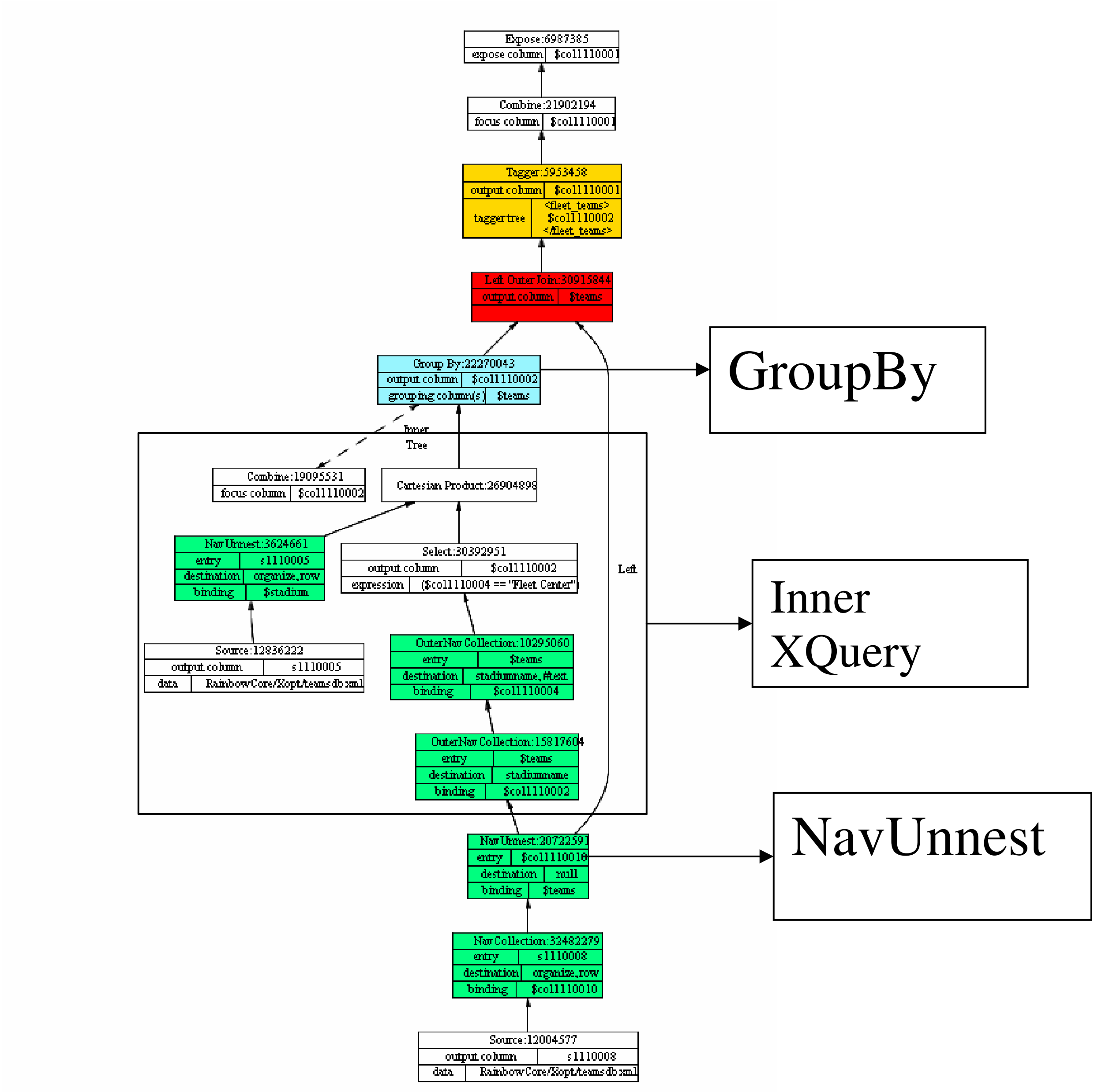

Expose:6987385
expose column | $col1110001
Combine:21902194
focus column | $col1110001
Tagger:5953458
output column | $col1110001
taggertree | <fleet_teams> $col1110002 </fleet_teams>
Left Outer Join:30915844
output column | $teams
Group By:22270043
output column | $col1110002
grouping column(s) | $teams
GroupBy
Inner Tree
Combine:19095531
focus column | $col1110002
Cartesian Product:26904898
NavUnnest:3624661
entry | s1110005
destination | organize,row
binding | $stadium
Select:30392951
output column | $col1110002
expression | ($col1110004 == "Fleet Center")
Left
Inner XQuery
Source:12836222
output column | s1110005
data | RainbowCore/Xopt/teamsdb.xml
OuterNav Collection:10295060
entry | $teams
destination | stadiumname, #text
binding | $col1110004
OuterNav Collection:15817604
entry | $teams
destination | stadiumname
binding | $col1110002
NavUnnest:20722591
entry | $col1110016
destination | null
binding | $teams
NavUnnest
Nav Collection:32482279
entry | s1110008
destination | organize,row
binding | $col1110010
Source:12004577
output column | s1110008
data | RainbowCore/Xopt/teamsdb.xml

Figure 5 Xopt 7: inner and outer (2)

Let us try to give it some theoretical explanation. As we have observed, only NavUnnest can have to paths up to its ancesters(a NavUnnest may have two fathers). One path serves the inner and the other path serves the outer. So if there is one filter we need only for the outer, we should put it on the path serving the outer. It also makes sense that when this NavUnnest serves the inner, usually it is used to filter out some tuples and according to "Push down" Policy, we need to make the filter near this NavUnnest as near as possible. But apparently these operators belong to the inner.

Having the ability to differentiate the inner XQuery and outer XQuery, we can set corresponding rules and detailed algorithm to realize it, if time allows.

10 Appendix C: Discussion about the policy of ignoring LET

In the rewrite tree, the hardest-to-find clause is the "Let clause". When we see NavCollection/OuterNavCollection, there may be a "Let clause". We can set a set of complex rules to make it appear in the final XQuery statement. But we decide to "ignore" the "Let clause".

Let us take the following figure as an example.

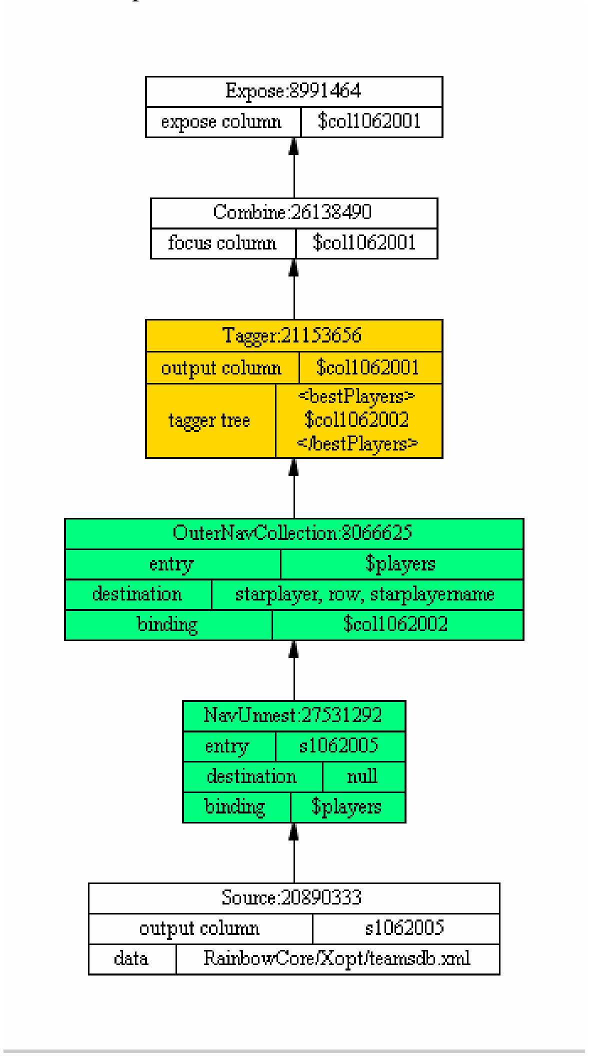

Expose:8991464
expose column | $col1062001
Combine:26138490
focus column | $col1062001
Tagger:21153656
output column | $col1062001
tagger tree | <bestPlayers> $col1062002 </bestPlayers>
OuterNavCollection:8066625
entry | $players
destination | starplayer, row, starplayername
binding | $col1062002
NavUnnest:27531292
entry | s1062005
destination | null
binding | $players
Source:20890333
output column | s1062005
data | RainbowCore/Xopt/teamsdb.xml

Figure 6 Rainbow Rewrite Tree Xopt-3

First we see Source and NavUnnest. Obviously this is a binding, corresponding to a For clause. Then OuterNavCollection will further navigate the variable and show the value in the Tagger Tree. It is quite natural that we can write down the XQuery statement as:

```
FOR $players IN document("RainbowCore/Xopt/teamsdb.xml")
RETURN
        <bestPlayers>
      $players/starplayer/row/starplayername
        </bestPlayers>
```

This is also the result of XATOX. However the original XQuery is:

```
FOR $players IN document("RainbowCore/Xopt/teamsdb.xml")
LET $playerName := $players/starplayer/row
RETURN
```

```
<bestPlayers>
$playerName/starplayername
</bestPlayers>
```

Is there any difference between the two XQueries? As we know, "Let clause" is doing the local binding: for the current value bound to the variable in "For Clause" (Now we only consider the cases that there is always a For clause, the situation that there is only a Let Clause will be handled separately and detailed discussed latter), use another variable to bind it and then give the result in one element. For this example, after we ignore Let, the XQuery statement means: for every $players, show its $players/starplayer/row/starplayername, and use <bestPlayer> Tagger to denote it. This means exactly the same as the original statement.

In order to prove this, we use our generated XQuery(without Let), feed it to the Rainbow and get the rewrite Tree, shown as follows. Compared with the figure 1, we find no difference.

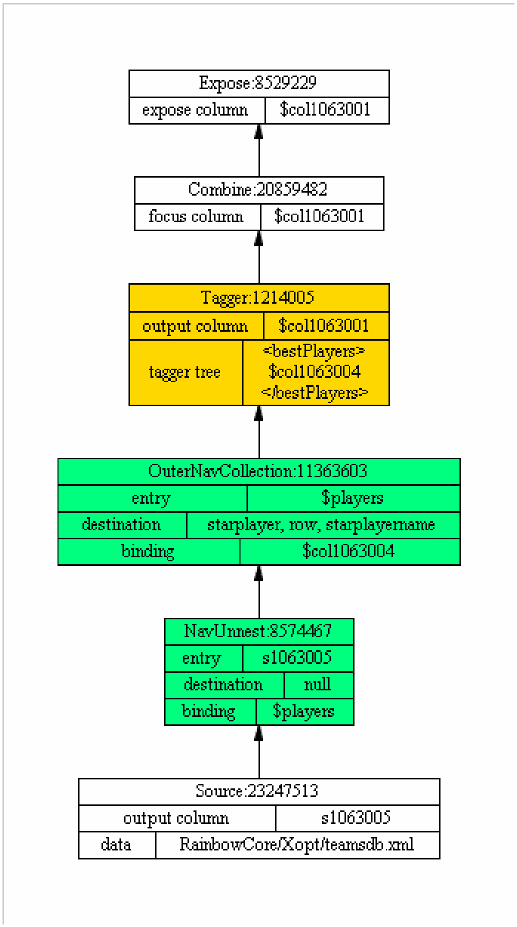


Figure 7 Rainbow Rewrite Tree of XATOX's result of Xopt-3

From this example, we further generalize what we find: Let clause is just like "#define" in C language. It will make the statement more readable (imagine the situation that we need to RETURN count (FLWR), use Let =FLWR will make the statement more clear. But we leave this to future discussion due to limited time), however it is just doing some pure "textual substitution" (or at least it is providing the same result), just like the "Laziness" of Scheme. So the Let Clause is set mostly for users, not for the express power or the interpreter.

Also, we should not forget that in some situations there is Let Clause without For Clause. This situation is quite easy to handle. Because there is no For Clause, there is no NavUnnest. So when we find Source + NavCollection and there is no more NavUnnest, this must correspond to a Let Clause.

However, since Let clause is somewhat unnecessary, there is no reason to handle this situation in this way, even if it is easy to implement. We continue our "ignoring Let" policy: simply do nothing.

Let us take another example:

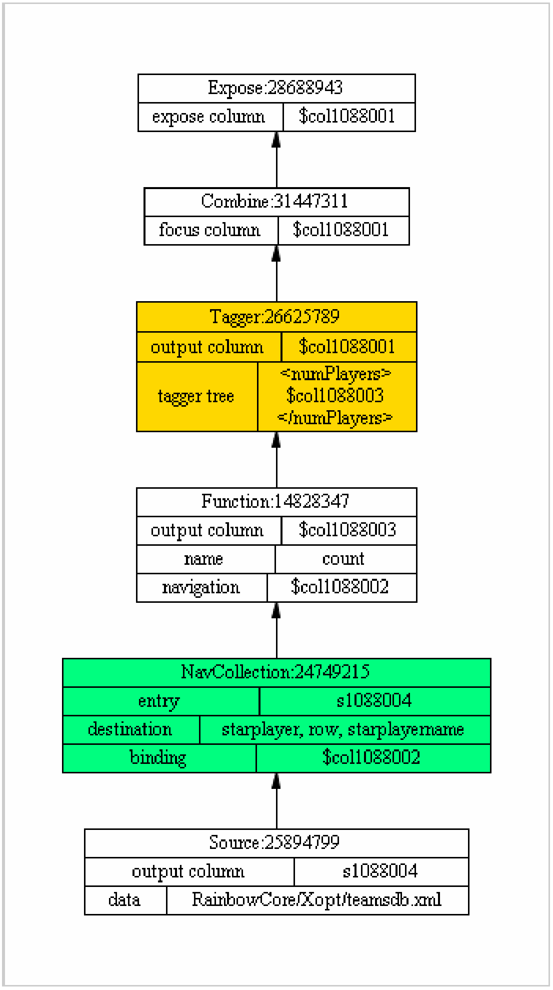


Figure 8 Rainbow Rewrite Tree Xopt 10

Obviously the result of XATOX will be:
RETURN <numPlayers>count(
document("RainbowCore/Xopt/teamsdb.xml")/starplayer/row/starplayername)</numPlayers>

And the original XQuery statement is:

```
LET $players := document("RainbowCore/Xopt/teamsdb.xml")/starplayer/row
RETURN
        <numPlayers>
        count($players/starplayername)
        </numPlayers>
```

Again they will have the same result. The following is the generated Rewrite Tree when we feed XATOX's result (without Let) to the Rainbow system.

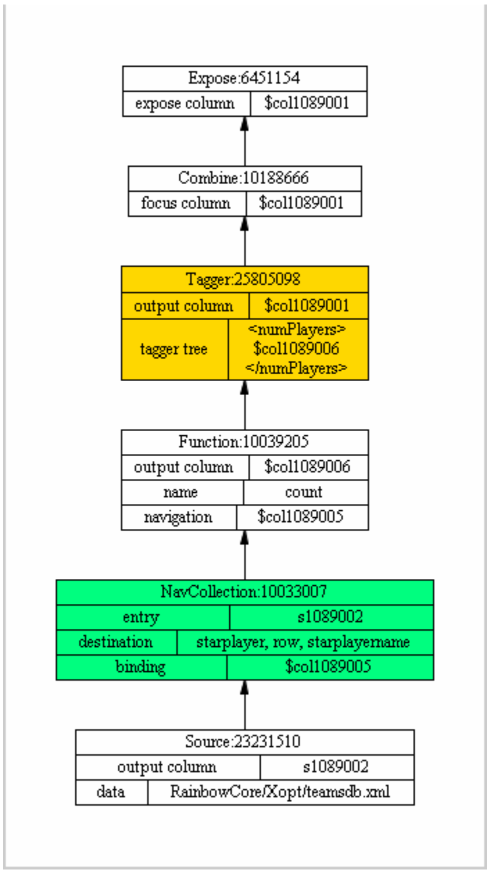
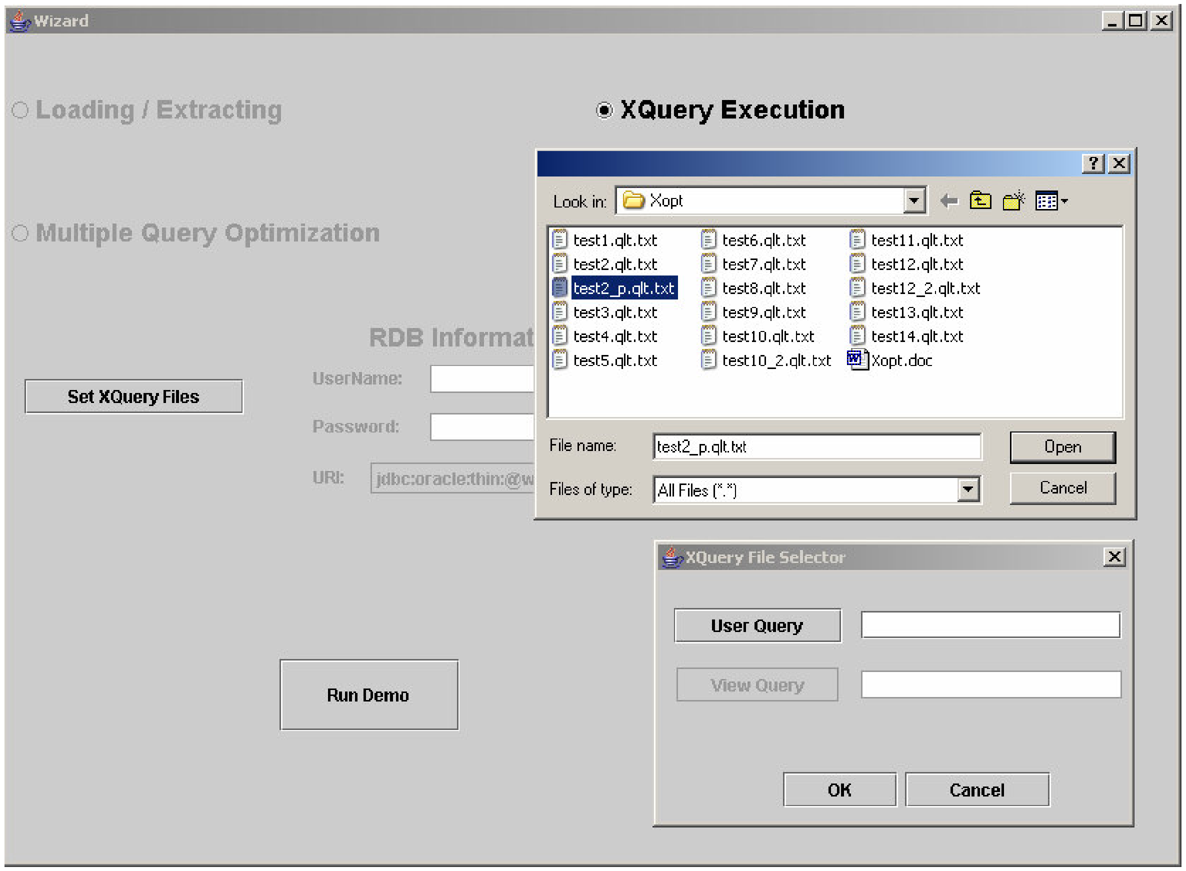


Figure 9 Rainbow Rewrite Tree of XATOX's result of Xopt-10

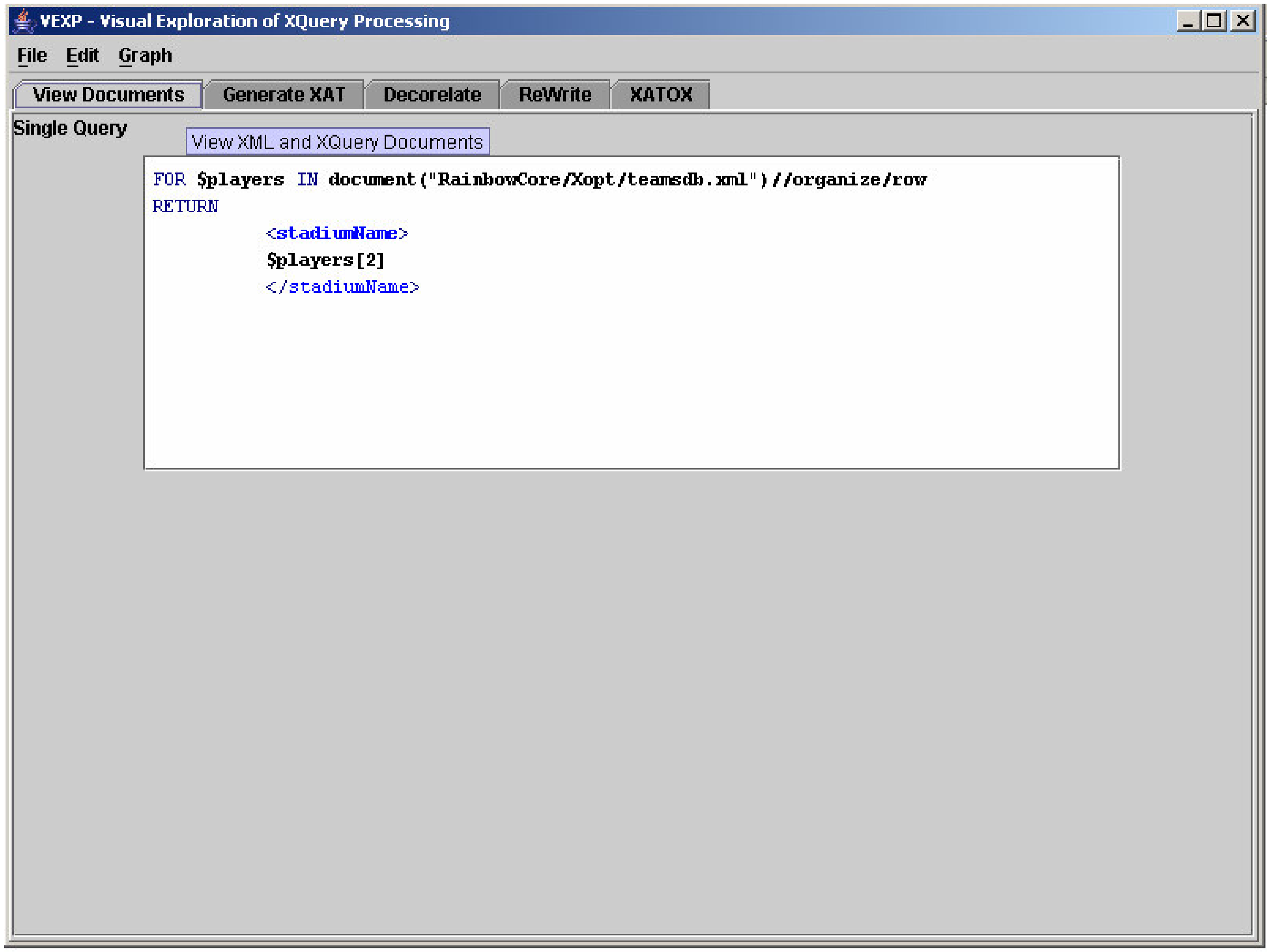
As we can see here, whether there are For clauses or not will not influence our policy of ignoring "Let Clause".

11 Appendix D: Sample Run

When running XATOX through Rainbow's interface we get the following GUI to indicate were the location of the user's XQuery file as follows:

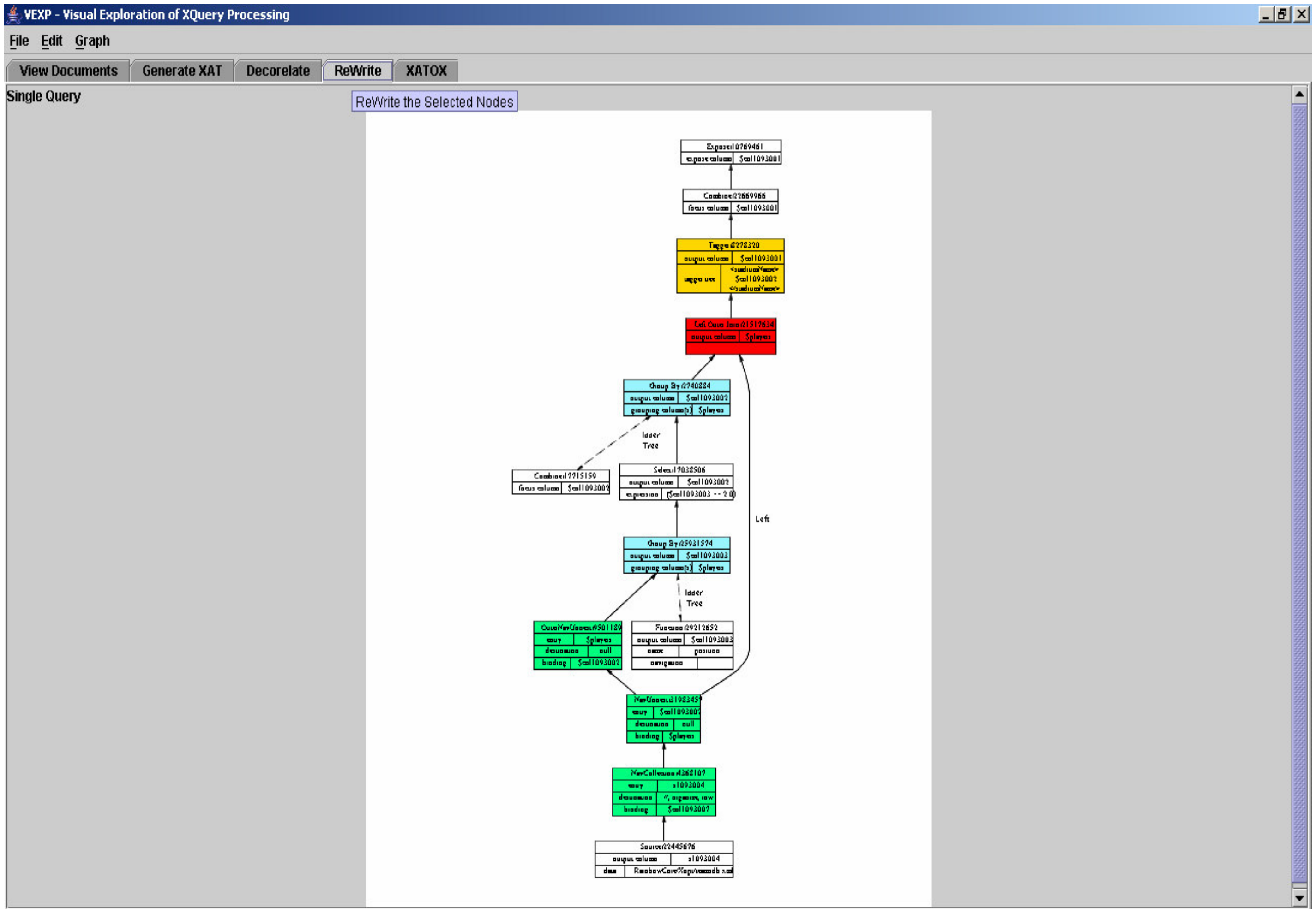


The first tab, “ViewDocuments”, will show us the user XQuery Statement selected from the Xopt Test Cases folder:


VEXP - Visual Exploration of XQuery Processing
File
Edit
Graph
View Documents
Generate XAT
Decorelate
ReWrite
XATOX
Single Query
View XML and XQuery Documents

```
FOR $players IN document("RainbowCore/Xopt/teamsdb.xml")//organize/row
RETURN
          <stadiumName>
          $players[2]
          </stadiumName>
```

The next two tabs we do not care for our project XATOX (Generate XAT and Decorelate), we only care about the Optimized XAT (Rewrite) tree presented by Rainbow as follows:



To be able to debug our code we present our XATOX Hash Table in the console as follows:

```
XATOX HASH TABLE
----------------

 __________  __     ______________________________     ____________________   ____________   _______________________________________
[IDENTIFIER][IO][T][____________PATH______________][F][________NAME________][___OUTPUT___][______________DEPENDENTS_______________]
{30283254__}{4_}{1}{[0.0.0.0., 0.0.0.1.0.0.0.0.]__}{0}{NavUnnest___________}{$players____}{[$col1094007]__________________________}
{11240282__}{7_}{1}{[0.0.0.1.0.]__________________}{0}{Select______________}{$col1094002_}{[$col1094003]__________________________}
{25798515__}{2_}{1}{[0.0.]________________________}{0}{Tagger______________}{$col1094001_}{[$col1094002]__________________________}
{5819561___}{1_}{1}{[0.]__________________________}{0}{Combine_____________}{null________}{[$col1094001]__________________________}
{22221117__}{6_}{1}{[0.0.0.0.0.]__________________}{0}{NavCollection_______}{$col1094007_}{[s1094004]_____________________________}
{2608483___}{5_}{1}{[0.0.0.1.]____________________}{0}{GroupBy_____________}{$col1094002_}{[$players]_____________________________}
{7718797___}{8_}{0}{[272782]______________________}{0}{position____________}{$col1094003_}{[]_____________________________________}
{18871377__}{9_}{1}{[0.0.0.1.0.0.0.]______________}{0}{OuterNavUnnest______}{$col1094002_}{[$players]_____________________________}
{272782____}{8_}{1}{[0.0.0.1.0.0.]________________}{0}{GroupBy_____________}{$col1094003_}{[$players]_____________________________}
{17051596__}{5_}{0}{[2608483]_____________________}{0}{Combine_____________}{null________}{[$col1094002]__________________________}
{3737099___}{7_}{1}{[0.0.0.0.0.0.]________________}{0}{Source______________}{s1094004____}{null___________________________________}
{22555260__}{3_}{1}{[0.0.0.]______________________}{0}{LOJ_________________}{$players____}{null___________________________________}
```

The PRE-VALUES are as follows:

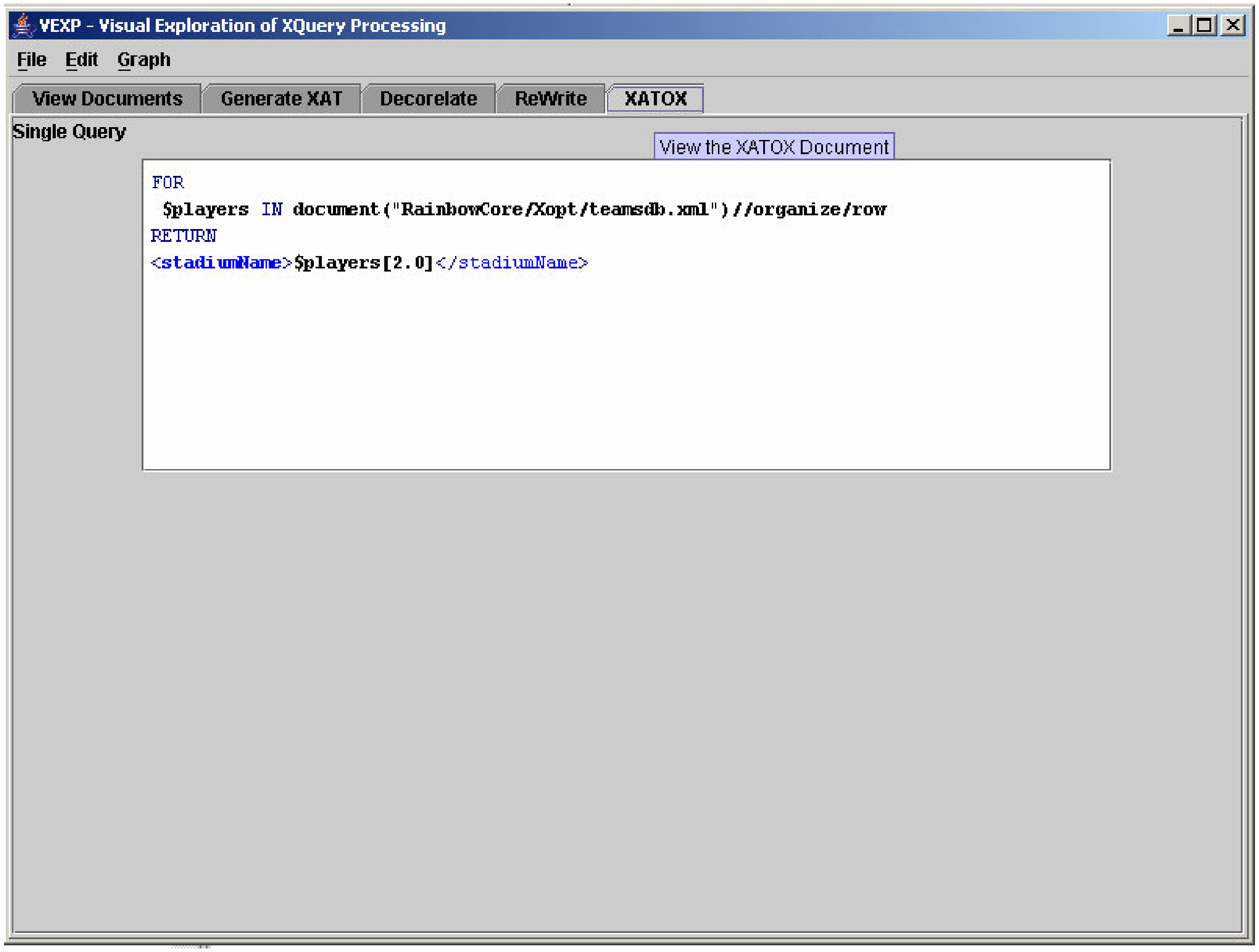
```
XATOX HASH TABLE
----------------

 ______________________________  ____________________  ______________________________________________________________
[____________PATH______________][________NAME________][_________________________PRE-VALUE____________________________]
{[0.0.0.0., 0.0.0.1.0.0.0.0.]__}{NavUnnest___________}{$col1098007->_________________________________________________}
{[0.0.0.1.0.]__________________}{Select______________}{($col1098003 == 2.0)__________________________________________}
{[0.0.]________________________}{Tagger______________}{<stadiumName>{$col1098002}</stadiumName>______________________}
{[0.]__________________________}{Combine_____________}{null__________________________________________________________}
{[0.0.0.0.0.]__________________}{NavCollection_______}{s1098004->////organize/row____________________________________}
{[0.0.0.1.]____________________}{GroupBy_____________}{[$players]:$col1098002________________________________________}
{[272782]______________________}{position____________}{null__________________________________________________________}
{[0.0.0.1.0.0.0.]______________}{OuterNavUnnest______}{$players->____________________________________________________}
{[0.0.0.1.0.0.]________________}{GroupBy_____________}{[$players]:$col1098003________________________________________}
{[2608483]_____________________}{Combine_____________}{null__________________________________________________________}
{[0.0.0.0.0.0.]________________}{Source______________}{RainbowCore/Xopt/teamsdb.xml__________________________________}
{[0.0.0.]______________________}{LOJ_________________}{null__________________________________________________________}
```

And the FINAL-VALUES are as follows:

```
XATOX HASH TABLE
----------------
 ______________________________  ____________________  ____________________________________________________________________________________________
[____________PATH______________][________NAME________][_______________________________________FINAL-VALUE_______________________________________]
{[0.0.0.]______________________}{LOJ_________________}{IGNORE THIS OPERATOR______________________________________________________________________}
{[0.0.0.0., 0.0.0.1.0.0.0.0.]__}{NavUnnest___________}{ $players IN document("RainbowCore/Xopt/teamsdb.xml")//organize/row_______________________}
{[0.0.0.0.0.]__________________}{NavCollection_______}{ document("RainbowCore/Xopt/teamsdb.xml")//organize/row_________________________________}
{[0.0.0.1.]____________________}{GroupBy_____________}{$players[2.0]____________________________________________________________________________}
{[5584739]_____________________}{Combine_____________}{null_____________________________________________________________________________________}
{[0.0.]________________________}{Tagger______________}{<stadiumName>$players[2.0]</stadiumName>__________________________________________________}
{[0.0.0.1.0.0.]________________}{GroupBy_____________}{position( $players IN document("RainbowCore/Xopt/teamsdb.xml")//organize/row)_____________}
{[0.0.0.1.0.0.0.]______________}{OuterNavUnnest______}{NOT YET__________________________________________________________________________________}
{[0.0.0.1.0.]__________________}{Select______________}{$players[2.0]____________________________________________________________________________}
{[18871377]____________________}{position____________}{null_____________________________________________________________________________________}
{[0.0.0.0.0.0.]________________}{Source______________}{ document("RainbowCore/Xopt/teamsdb.xml")_________________________________________________}
{[0.]__________________________}{Combine_____________}{FOR  $players IN document("RainbowCore/Xopt/teamsdb.xml")//organize/row RETURN
<stadiumName>$players[2.0]</stadiumName> }
```

Note that the row with the operator's name equal to Combine has the result of our XATOX translation. That is because it is the node right below the root (Expose) in the Rainbow's XAT Tree. So the node that is the son of the root of the XAT Tree will always contain the result in the end of the execution and save it into a predefined file (XATOX.txt).

The last tab (XATOX), will read the file we created and display its results in Rainbow's GUI as follows:



```
FOR
 $players IN document("RainbowCore/Xopt/teamsdb.xml")//organize/row
RETURN
<stadiumName>$players[2.0]</stadiumName>
```