

# TGV : a Tree Graph View for Modelling Untyped XQuery

Nicolas TRAVERS<sup>1</sup>

Tuyêt Trâm DANG NGOC<sup>2</sup>

Tianxiao LIU<sup>2,3</sup>

<sup>1</sup>PRiSM Laboratory  
University of Versailles  
France

<sup>2</sup>ETIS Laboratory  
University of Cergy-Pontoise  
France

<sup>3</sup>XCalia S.A.  
Data&service intermediation  
France

<sup>1</sup> {firstname.lastname}@prism.uvsq.fr

<sup>2</sup>{firstname.lastname}@u-cergy.fr

## Abstract

XML [7] has become a de facto standard to exchange and represent any kind of data in various contexts. XML data can be manipulated using the XQuery [31] language, which can express though a compact and comprehensive way any queries and transformations. An XQuery expression is evaluated as follows. (1) the expression is rewritten into a "canonical XQuery", then (2) it is modeled in an internal representation and finally (3) it is optimized and executed.

In [9], Chen et al. proposed to rewrite XQuery expressions in "canonical XQuery". However, their solution is restricted to a rather limited subset of XQuery and does not support complex XQuery expressions. [4] introduced the TPQ model to represent a single FWR statements as a Tree Pattern and a formula. Then [9] proposed the GTP model which generalize their approach to support several FWR. However, their model cannot capture well all the expressiveness of XQuery expressions, cannot handle mediation problems and do not support extensible optimizations.

In our paper, we made three contributions. First, we extend the rules developed by [9] to rewrite any XQuery expression into a "canonical XQuery". Second, we design a new model called TGV which supports all the functionalities of XQuery, uses an intuitive representation compliant with mediation issues, and provides a support for optimization and cost information. Finally, we provide a generic cost model coupled with the TGV.

**Keywords:** XQuery evaluation, TGV, Extensible optimization, Cost model

# 1 Introduction

Data sources on the Internet are many and varied. To integrate data coming from distributed and heterogeneous sources, the famous mediators/wrappers architecture [33] is generally used. In this architecture, the user issues a request to the mediator which sends in turn part of the request to wrappers associated to data sources. The result is then sent back to the mediator which integrates the result accordingly. In order to guarantee interoperability between the different elements of the architecture, we consider XML for data representation and XQuery as the query language.

Indeed, XML [7], has become the preferred way to represent semi-structured data [1] which has become an effective way to define any type of data that can be represented as a tree.

Moreover, XQuery [31] has proved to be an expressive and powerful query language to query XML data both on structure and content, and to make transformation on the data. In addition, its query functionalities come from both the database community (filtering, join, selection, aggregation), and the text community (supporting and defining function as text search).

However, the complexity of the XQuery language makes its evaluation very difficult. To alleviate this problem, most of the systems support only a limited subset of the XQuery language.

[9] has shown that some forms of XQuery expressions could be rewritten to one generic form called "canonized XQuery". Rules that transform an XQuery expression to an equivalent expression with a generic form (said "canonized") have also been demonstrated. We will see in section 2.1.1 that [9] only supports a subset of XQuery formulation and do not deal with all type of XQuery expressions (set, conditional, sequential, scheduling etc.) The first contribution of our paper is to extend the work of [9] in order to canonize any XQuery expressions.

The canonized XQuery expressions require a logical structure model to be manipulated, optimized and then evaluated. [4] has introduced the TPQ model that expresses a single *FWR* query by a Pattern Tree and a formula. Then, [9] proposes GTPs that are a generalisation of TPQs in which each *let* clause generates a Tree Pattern, and the formula contains all the operations. The representation is intuitive and acts as a template for the

data source, as QBE [37] did for querying relational data. However, GTPs do not capture well all the expressivity of XQuery, cannot handle mediation problems, and do not support extensible optimisation. To solve these problems, we made a second contribution. We build a new model called TGV which provides the following features: (a) we integrate the whole functionalities of XQuery (collection, XPath, predicate, aggregate, conditional part, etc.) (b) we use an intuitive representation that provides a global visualisation of the request for the mediation. (c) we provide a support for optimisation and a support for evaluation information (e.g., cost, sources location).

Finally, our third contribution is to propose optimizations to improve query evaluation. To this end, we define transformation rules to generate a set of equivalent TGV (giving the same results) from the same query. The choice of the best TGV is based on specific costs (response time, money cost, bandwidth, ...). A support for cost models is also provided by a new language based on MathML which can consider different sources distribution and their capabilities.

We have implemented the whole XQuery process in our mediation system XLive [12]. Figure 1 describes the evaluation process: (1) XQuery is canonized into a canonical form of XQuery (2) then the canonized XQuery is modeled in the internal structure TGV which can (3) be restructured into equivalent structures using equivalence rules. (4) Then the TGV is annotated with information for evaluation such as the data sources location, cost models information, sources functional capabilities, etc. The optimal annotated TGV is then selected and (5) the logical TGV is transformed into an execution plan using a physical algebra. We have chosen the XAlgebra [11], that is an extension of the relational algebra to XML. (6) Finally, the execution plan is evaluated and produces an XML result.

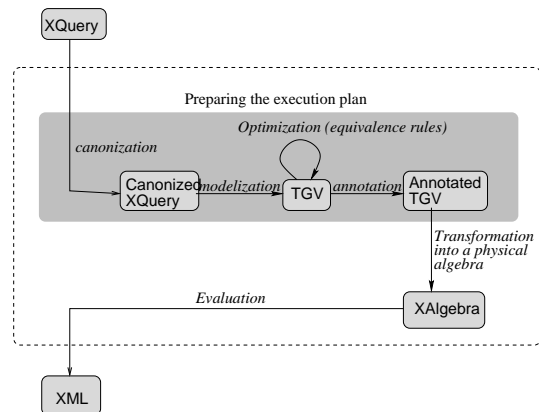


Figure 1: Evaluation processing

The whole process is implemented in the XLive [12] system and validate all use-cases defined by the W3C that do not implies strong typing consideration (our system recognize 8 of the 9 categories of XQuery).

In this paper we describe the preparation of the XQuery evaluation. This article is organized as follows. The next section focuses on canonizing XQuery to restrict the study. Then, section 2 introduces the TGV structure that we had defined for modeling XQuery in a practical way. In section 3 we define some rules to manipulate the TGV, we show in section 4 how the TGV can be annotated with extra information to support sources description and evaluation cost estimation (subsection 4). Finally section 5 concludes.

## 2 XQuery Modeling

### 2.1 Canonization

XQuery is a rich and complex language. Its powerful expression capabilities provides a large range of queries over XML documents. However the richness of the language makes the evaluation of XQueries inefficient in several cases. In order to optimize the evaluation of such queries, we propose to structure the language declaration by canonizing it using some transformation rules. These transformation rules keep the semantic of the query and make them more convenient to manipulate.

#### 2.1.1 Canonization (existing work)

[9] proposes rules of canonization to translate XQueries to their single form. [9] introduces the following theorems to transform XQuery queries : (a) *Moving XPath filters to where clauses*: The XPath filters are applied to a specific node, like an operation in the *where* clause. A transformation rule translates this filter in a new query where the filter operation is moved into the *where* clause. The XPath is split in appropriate paths to generate an equivalent query. (b) *unnesting return clauses*: Each FLWR queries nested in the *return* clause is redefined in a *let* clause and is assigned to a variable. In the *return* clause, the nested queries are

replaced by their assigned variable . (c) *isolating aggregate functions*: Each aggregate function is redefined in a *let* clause and is assigned to a variable which replaces the aggregate function in the main expression. (d) *isolating quantifier functions*: Each quantifier functions (*every and some*) is redefined in a *let* clause and is assigned to a variable which replaces the quantifier functions in the main expression.

XQuery queries can be canonized in a simpler form using previous translation rules. However the lack of ordering operators (*order by*), set operators (*union, except, intersect*), conditional operators (*if/then/else*) and sequential expressions (with parenthesis) reduce tremendously the XQuery expressiveness. We propose in the next section to extend canonization of XQuery.

#### 2.1.2 Extension

We extend the XQuery definition domain by defining new additional canonization rules. These rules (summarized in table 1) have been demonstrated in [29] and are as follows :

- R1: Each of the *Order by* clauses are declared in a *let* clause. The ordering is defined by a specific function *orderby()*. This rule is an extension of the transformation rule *isolating aggregate functions* defined in 2.1.1(c).
- R2: Two FLWOR expressions connected by a set operator are declared in a *let* clause and are each assigned to a variable. The result - which is a set operation between the two previous variables - is assigned to a variable.
- R3: Nested expressions in conditional expressions *If/Then/Else* are declared in a *let* clause and are each assigned to a variable. The conditional expression is rewritten using variables previously defined.
- R4: Each sequential expressions enclosed in parentheses on a set of values or XPaths is transformed into a *let* clause and is assigned to a variable.

We have defined the rules to get a specific ordered XQuery expression. These rules are summed up in the table 1.

	Expressions	Canonical form
R1	$order\ by\ expr_1$	$\Rightarrow let\ \$l_1 := orderby(expr_1)$
R2	$(expr_1 \mid expr_2)$	$\Rightarrow let\ \$l_1 := expr1, \$l_2 := expr2, \$l_3 := (\$l_1 \mid \$l_2)$
	$(expr_1 \ \& \ expr_2)$	$\Rightarrow let\ \$l_1 := expr1, \$l_2 := expr2, \$l_3 := (\$l_1 \ \& \ \$l_2)$
	$(expr_1 - expr_2)$	$\Rightarrow let\ \$l_1 := expr1, \$l_2 := expr2, \$l_3 := (\$l_1 - \$l_2)$
R3	$if\ expr_1\ then\ expr_2\ else\ expr_3$	$\Rightarrow let\ \$l_1 := expr_2, \$l_2 := expr_3$ $if\ expr_1\ then\ \$l_1\ else\ \$l_2$ (if $expr_2$ and $expr_3$ are nested queries)
R4	$(expr_1)/expr_2$	$\Rightarrow let\ \$l_1 := expr_1$ $\$l_1/expr_2$

Table 1: XQuery canonization rules

The validation of these rules can be found in [29]. Using these rules, we obtain a canonical XQuery expression that is ready for a translation to our query internal representation, based on set of trees and external relations between them.

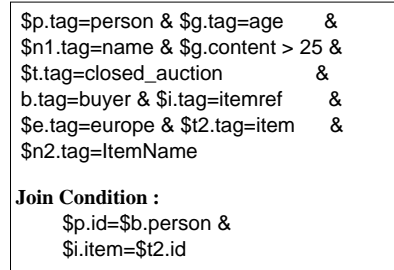
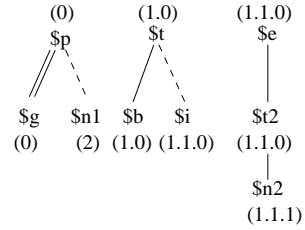


Figure 2: Generalized Tree Pattern (GTP)

However, GTP does not model all constructs that are required for distributed queries:

- The notion of data source is not included.
- The XML result of a query is not modeled.
- Modeling views and query on views are not possible.
- Let and functions are not integrated.
- Tags, relations, and constraints are embedded in a boolean formula difficult to read.

## 2.2 Tree Graph Views (TGV)

Although canonization reduces the XQuery queries expression to a canonical form, XQuery modeling is a difficult goal since XQuery provides a large set of functionalities. Moreover, mediation purpose directs our reflection for modeling in specific Tree Patterns in which all XQuery operations are represented in a single expressive form. Each set of trees defined in canonized queries generates a particular representation.

A well-known structure for modeling XML queries is the so-called Tree Pattern Query (TPQ). A TPQ is a tree with nodes labeled by variables together with a formula specifying constraints on tags, attributes, and contents. The tree consists in two kinds of edges parent-child (pc) and ancestor-descendant (ad) edges. The semantics of a TPQ is based on the pattern matching notion, a mapping from the TPQ nodes to the database nodes satisfying the formula. A Generalized Tree Pattern is a natural extension of TPQs. XQuery contains joins, nesting, aggregates and other complex constructs not captured by TPQ. GTP integrates most of them, including optional and mandatory nodes. An example of a GTP is given in Figure 2. Notice the dotted line, the join conditions, and the block levels (e.g., 1.1.0). Block levels identify links between TPQs and are defined by the hierarchy level in the group number.

All in all, the GTP representation is not very intuitive, as dependencies between tree patterns do not appear except in formulas. And there is no support for additional information useful at evaluation time (cost, sources location). Thus, this model requires some extensions and adaptations to be the core of a distributed query-processing algorithm in a mediator. We propose the TGV (Tree Graph View) model that we have implemented in the XLive mediator for XQuery processing.

We now see all the characteristics of the Tree Graph View model. First, we introduce *TreePatterns* which are the XML document filters, and specific structures adapted to XQuery requirements. Then, *Constraints* are added to this model to integrate general filters, which can be attached to any type of the model. To complete this model, *Hyperlinks* are introduced to link together preceding structures. A Tree Graph View is composed of all this structure to model a complete XQuery query.

### 2.2.1 Tree Pattern

A *Tree Pattern* is a tree with different tags an XML document must match with. This template is a set of *XPaths* extracted from the XQuery query. In the Tree Pattern representation, each tag of the XPath becomes a *Node*. Every *Nodes* are linked together by *Node Links* representing hierarchical information between two *Nodes*. We now give definitions of each Abstract Data Type (ADT) of a Tree Pattern (Figure 3) :

- **Node** : A *Node* is the tag name that an element in an XML document must have to be relevant to this Tree Pattern.  
*A node corresponds to an element in an XML document. Each one that match with the given Node name is relevant to the given query. To introduce specific node names, nodes can have namespace and wildcard definitions.*
- **NodeLink** : A *Node Link* is a hierarchical link between two *Nodes*. It contains three information: parent/child or ancestor/descendant, mandatory or optional, and the declaration place in the query.  
*A Node Link represents the relation between two nodes. Then, two elements of an XML document must respect this relation to be relevant to the given query. When a link has a parent/child information, elements corresponding to the second node must be a direct child of the first one. Contrary to a descendant/ancestor link in which the second node can be any descendant element contained into the first one. Concerning a mandatory link, the following node must appear in the XML document to be relevant to the query, then this information is said to be discriminant to the query. On the opposite, an optional node link is not discriminant for the query. So, this node does not have a major influence on the*

*built XML result set. The declaration place information can be used in optimization purposes to replace nodes requirements (it isn't an ordered information between nodes)*

A set of *Nodes* and *Node Links* generates a *Tree Pattern* that filters XML trees the given patterns must match with.

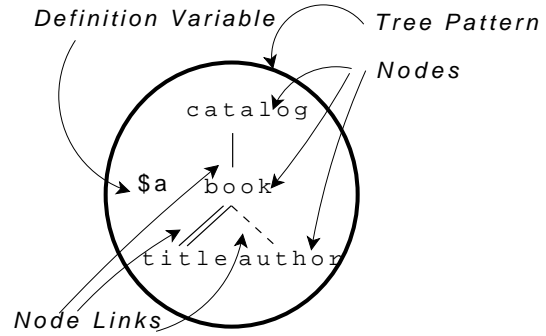


Figure 3: Representation of a Tree Pattern

As we can see in figure 3, *Nodes* and *Node Links* are associated together to form a *Tree Pattern* that is a filter representation on an XML document. We can identify different types of node links between nodes. In fact, links between *catalog/book* and *book/author* have a parent-child status since it is represented by a single link, contrary to the *book/title* link that have a ancestor-descendant status from the double link. Moreover, we can define the optional status by dotted links like between *book* and *author*, others in plain line are mandatory links.

#### Definition 1 : TreePattern

A *Tree Pattern* is a set of *Nodes* and *Node Links* with a definition variable. It represents a tree pattern that an XML document must match with the specific element associations to be relevant for a given query.

To represent all XQuery richness, we must introduce specific *Tree Patterns* to model each characteristic of XQuery:

- **SourceTreePattern** : A *Source Tree Pattern* is a *Tree Pattern* defined by a targeted document or set of documents and a root path.  
*A Source Tree Pattern corresponds to a for declaration on a targeted XML document with a specific root path, that defines the set of trees to work with.*

- **IntermediateTreePattern** : An *Intermediate Tree Pattern* is a *Tree Pattern* defined on a previous one that specializes the domain on a specific Node.

An *Intermediate Tree Pattern* corresponds to a *for* declaration that specializes an element by creating a new set of trees. Thus, it creates a *Tree Pattern* that defines a new domain.

- **ReturnTreePattern** : A *Return Tree Pattern* is a *Tree Pattern* that defines the result construction of an XML document. Each node coincide with a tag on the XML result set, or a contents text.

A *Return Tree Pattern* corresponds to the *return* clause of an *XQuery* query, that builds the main XML resulted document. We can identify tags by nodes, attributes are preceded by a "@", and text are only a quoted text. Required *XPath* are treated by hyperlinks (see below) as an empty child node.

- **AggregateTreePattern** : An *Aggregate Tree Pattern* is a *Tree Pattern* that builds a temporary result set. It can represent nested queries and aggregated functions on a set of trees.

An *Aggregate Tree Pattern* corresponds to a *let* clause that defines a treatment on a set of trees. By canonization rules, we can see that nested queries are redefined in those clauses, so they build a temporary result set which is projected (see hyperlinks below) into this *Aggregate Tree Pattern*. *Aggregate Functions* are applied on a set of trees, then we must represent them by this model.

We introduced *Tree Patterns* that can define all required model of pattern matching on trees. Then, when an XML document is relevant to our *Tree Pattern*, it is selected by the model. But, since *XQuery* queries are more complicated than a simple *Tree Pattern*, we need to add some other types to our model.

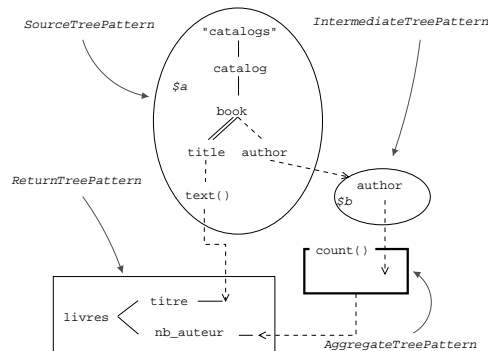


Figure 4: Different types of Tree Patterns

As we can see in figure 4, the four types of patterns trees are represented. On the top left, a *SourceTreePattern* defines the set of selected trees for the TGV. Then, an *IntermediateTreePattern* on top right determines a new set of values, we can infer that it is optional since *NodeLink* (book/author) and the link between *author* and the ITP (called *SpecializedHyperlink*, see below) are optional. This *TreePattern* is projected into an *AggregateTreePattern* on the bottom right, we can identify the aggregate function *count* on *authors*, then we know that it defines the number of authors for a book. To finish, a *ReturnTreePattern* is defined on the bottom left to build a new XML document with the given tags, *title* and the number of *authors* are projected in this *Tree Pattern*. This *Tree Graph View* corresponds to the following *XQuery* query:

```
for $a in collection("catalogs")/catalog/book
where
  exists ($a//title)
return
  <books>
    <title>{$a//title/text()}</title>
    <nb_author>
      {for $b in $a/author
       return count ($b)}
    </nb_author>
  </books>
```

### 2.2.2 Constraints

In *XQuery* queries, constraints may be declared on an *XPath* to reduce the selectivity of a set of trees. Moreover, this constraints can be applied to a variable, like in *let* clauses, or between two *XPaths* to express join constraints between two set of trees. Thus, we introduce the type *Constraint* for this purpose.

#### Definition 2 : Constraint

A *Constraint* is a restriction of the feasible solutions in sets of trees. It can be applied to *Nodes*, *Tree Patterns*, *Hyperlinks*, *Constraints* or *Constants*. It appears as *Predicates* or *Functions*.

- **Predicate** : A *Predicate* is a constraint with a comparison operator between two links to different element types.

*Linked types* can be a constants, nodes, tree

patterns, hyperlinks or an other constraint in order to compose constraints.

- **Function** : A *Function* is a constraint with a name and a set of links to different element types.  
The function name defines the type of operation to treat. Linked types can be constants, nodes, tree patterns, hyperlinks or an other constraint for function composition.

Constraint representation depends of the linked element type. For a node, we put the constraint under the tag as we can see on figure 5. For a tree pattern, it is represented above it, as we saw the *count* function in figure 4 on the *AggregateTreePattern*. For hyperlinks, it depends of its type, as we will see on *JoinHyperlinks* in figure 5 a link between two nodes is annotated with a equality constraint. For constraint composition, we compose naturally at the position of the linked element (node, tree pattern, hyperlink).

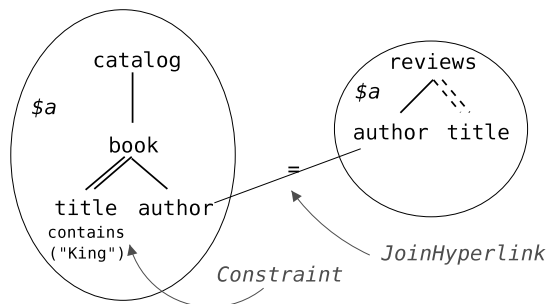


Figure 5: Example of node constraint

### 2.2.3 Hyperlinks

As XQuery queries can associate some variables together by join constraints, define new variables and *let* clauses, we introduce *Hyperlinks* as a link between types into our model. Thus, XQuery complexity generates few types of Hyperlinks in order to represent this expressiveness.

- **Hyperlink** : An *Hyperlink* is a link between two or more elements of a *Tree Graph View*. It represents an association by *Association Hyperlinks* or a transformation by *Directional Hyperlinks*.  
As we can see, an *Hyperlink* can be an *Association Hyperlink* for restriction purposes on some sets of elements, and a *Directional Hyperlink* to transform set of trees into a new one.

- **AssociationHyperlink** : An *Association Hyperlink* is an *Hyperlink* that connects two elements of the same type to represent a specific association, in order to filter results by verifying this association.

We can distinguish two types of *Association Hyperlinks*. *Join Hyperlinks* are made to connect two *Nodes* with a *Constraint* that represents a join constraint. *Constraint Hyperlinks* connect two *Constraints* with a *Boolean connector* associated to a return clause, in order to preserve constraint association (with *and/or* operators) and the treatment declaration level.

- **JoinHyperlink** : A *Join Hyperlink* is an association between two *Nodes* under *Constraint*. Relevant trees are those who verify this constraint with values given by nodes.

*Join Hyperlinks* are declared into *where* clauses to define a join constraint between two *XPaths*. Then it generates two nodes into *TreePatterns* which are connected together by this *Association Hyperlink* with the given constraint.

- **ConstraintHyperlink** : A *Constraint Hyperlink* is an association between two *Constraints* with a *Boolean connector*. It forms a list connected to a *Return-TreePattern* in order to keep treatment declaration level of this constraints. Relevant trees are those who verify all the connected constraints by *and/or* connectors, at a given declaration level.

A *Constraint Hyperlink* is a link between two constraints keeping a boolean operation status given by *where* clauses. It permits to know how processing constraints in the model. Furthermore, each nested query have a single return clause, thus we can define a treatment level associated to this clause. In fact, some constraints are not associated at the same level of their definition variable. Finally, aggregate functions can be declared in the *where* clause, its treatment is then associated to the *Constraint Hyperlink*.

- **DirectionalHyperlink** : A *Directional Hyperlink* is an injected transformation between elements. It specifies a transformation from a set of elements to a single one.  
*Directional Hyperlinks* are injected hyperlinks

since the destination element must be targeted only once.

- **ProjectionHyperlink:** : A *Projection Hyperlink* is a *Node to Node Directional Hyperlink*. It contains a mandatory or optional status. It represents a value projection of the given node into the projected one.  
*In XQuery queries, most projected nodes are those which are declared into return clauses. Some of them are built by XPath declarations into where or order by clauses. Mandatory status restricts results when the given node do not contains any values. It is deduced when it is declared into where and order by clauses. The Projection Hyperlink is set to be optional when a node is declared into the return clause.*
- **SpecializedHyperlink:** : A *Specialized Hyperlink* is a *Node to Tree Pattern Directional Hyperlink*. It contains a mandatory or optional status. It represents the specialization of a Node, by specifying a new TreePattern which root is the given node.  
*A Specialized Hyperlink permits to link a TreePattern with an IntermediateTreePattern. ITPs are declared in for clauses with a variable declaration on an existing one. This hyperlinks represent the variable association to the given node. The new TreePattern specialises the nodes association at this declaration level. This hyperlink can be optional since for clauses can be nested.*
- **GeneralizedHyperlink:** : A *Generalized Hyperlink* is a *Tree Pattern to Node Directional Hyperlink*. It contains a mandatory or optional status. It represents a TreePattern generalization result set, which result is projected into the given node.  
*A Generalized Hyperlink links a TreePattern with a node in order to represent aggregate projections. In fact, a set of trees built by an AggregateTreePattern is projected into a node. This hyperlink can have a mandatory status when the aggregate is declared into where clauses, else it has a optional status.*
- **SetHyperlink:** : An *Set Hyperlink* is a set of *Tree Patterns to Node under Con-*

*straint Directional Hyperlink*. The Constraint possible values are: Union, Intersect or Difference. It represents an set operation between few TreePatterns on a single Node.

*An Set Hyperlink represents the link between TreePatterns which result set is deduced by the set operation. This result set is projected into a given node on which we proceed further treatments.*

- **IfThenElseHyperlink:** : An *IfThenElse Hyperlink* is a set of *Elements to Node under Constraint Directional Hyperlink*. Elements can be a Node or an *AggregateTreePattern*, and the constraint is a Predicate or a Function. It represents a conditional expression which result is deduced by the constraint status.  
*Since XQuery is an operational language, we must associate this notion to our model. then IfThenElse Hyperlinks represent operational treatments based on results of the given constraint. We can project a Node (from an XPath) or an AggregateTreePattern (from a nested query).*

To represent these different hyperlinks, we use arrows between elements. *Projection Hyperlinks* is a single arrow between two nodes (dotted if optional status), an example is saw in figure 4 between *text()* and *title*, and between *author* and root of the *Aggregate Tree Pattern*. *Specialized Hyperlinks* is a single arrow between a node and a tree pattern (dotted if optional), in figure 4 it can be seen between the node *author* and the *Intermediate Tree Pattern \$b*. *Generalized Hyperlinks* link a tree pattern to a node, it is represented by a single arrow (dotted if optional), an example can be seen in figure 4 between the *count Aggregate Tree Pattern*, and *nb\_author* node.

For *Set Hyperlinks*, we need to introduce a multi-arrows and an set constraint. Each arrow comes from a tree pattern to link with others with the constraint. We can see it in the top of the figure 6 where two tree patterns (*\$a\_1* and *\$a\_2*) are linked to a *union* constraint, result is projected into the tree pattern *\$a*.

*IfThenElse Hyperlinks* needs three arrows (If, Then and Else clauses). The *if* clause is linked to a constraint, *then* and *else* clauses are linked to a node or a tree pattern. An example is given in the

bottom of the figure 6, where the *IfThenElse Hyperlink* give a *author* result (*then*) for *titles* containing the word "King" (*If*) or else give a string result "No King in title" (*else*).

We can see in figure 6 the corresponding XQuery query to the model on top-right. We can see the set operator "|" in the for clause for merging "catalogs" and "reviews" into the \$a variable. A conditional operator "if/then/else" is defined in the return clause in order to project the *author* value if the word "King" is present into the *title*.

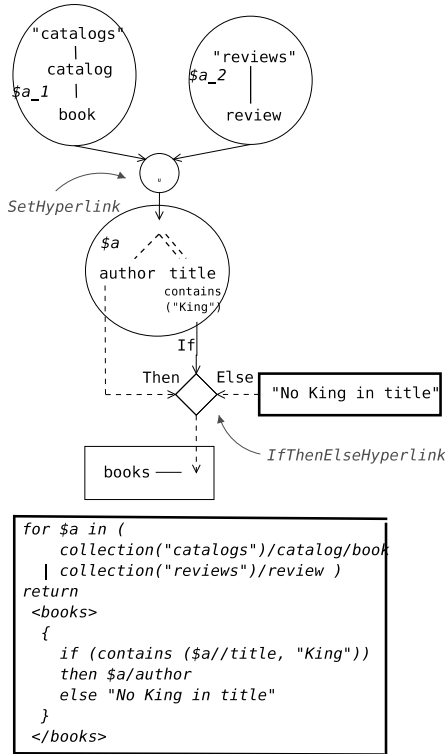


Figure 6: SetHyperlinks and IfThenElseHyperlinks

*Association Hyperlinks* are represented by a link between the associated *Return Tree Pattern* and constraints. This link is dotted if the boolean operator is typed "or". For visibility purposes, we will only represent this hyperlinks on graphs when a dotted operator is present or the level application of a constraint is different to the Tree Pattern's level.

Then, we define all *TreeGraphView* specific elements in order to represent XQuery expressiveness in a single model. All elements are linked together by hyperlinks representing specific associations given by XQuery specification. XML reconstruction is given by *ReturnTreePattern*, definition domains are set into *SourceTreePattern* and *IntermediateTreePattern*, aggregate operations are rep-

resented by *AggregateTreePattern*. We now define the *TreeGraphView* type in order to gather all this types into a single model.

### 2.2.4 Tree Graph Views

A *Tree Graph View* (TGV) is a representation of an XQuery query containing *TreePatterns*, *Constraints* and *Hyperlinks*. Input of the TGV is given by *SourceTreePatterns*, the output is defined by the *ReturnTreePattern* (not a *AggregateTreePattern* by inheritance).

#### Definition 3 : TreeGraphView:

A *Tree Graph View* is a set of *TreePatterns* linked together by *Hyperlinks*, and restricted by *Constraints*. It contains a set of *SourceTreePatterns* for inputs, a single *ReturnTreePattern* for the output.

Then we can model a canonized XQuery query with a *Tree Graph View* by transforming each clause by the associated type. Those types can generate hyperlinks in order to build a graph. This graph is a view of the query which can be transformed into an algebra to process it on XML documents.

### 2.2.5 Functions

In order to complete the model with XQuery requirements, we need to introduce function declarations. This expression take some elements in parameters and give a single element in return. Into our model, we will treat only parameters with the type *element()*.

Since parameters can modify few types into the XQuery query, we will use the same representation into *Tree Graph Views*. Thus, this parameters will be introduce into an *AggregateTreePattern* named by this declaration function with its variables definition. This variables will be find into the TGV by its name and *Projection Hyperlinks* between the *Aggregate Tree Pattern* and variables.

For treatment purposes, we will merge the TGV corresponding to the declaration function with the using function TGV only if necessary. In fact, recursive functions can not be represented since an infinite *Tree Graph View* will be generated.

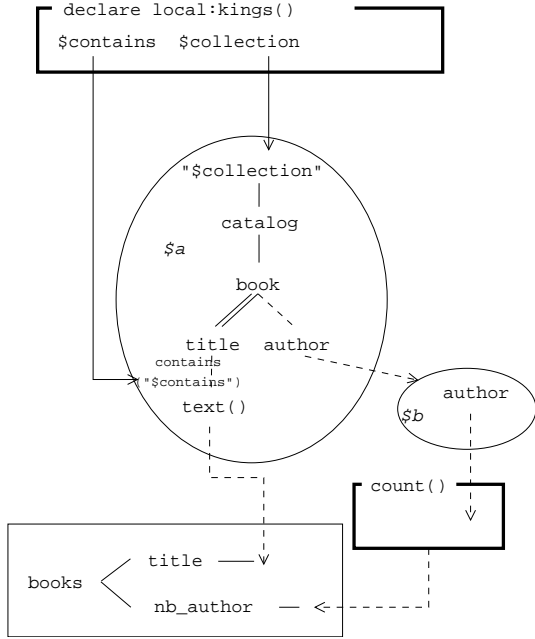


Figure 7: Example of declaration function

Figure 7 illustrates a declaration function *local:kings()* that bring two parameters into a Tree Graph View on the constraint function *contains* and the collection name of a *SourceTreePattern*. The function result is the last *Return Tree Pattern* declared.

### 3 Evaluation and Optimization

The previous section defined rules that translate XQuery expressions into a canonical form, and the TGV abstract data types used for modeling this canonized XQuery. The TGV model is a logical representation of XQueries. It is designed to be translated into an XML physical algebra to make evaluation of XML documents effective.

To compute the result of a TGV applied to XML documents, we need to define an evaluation operator for the TGV and also for each of its component. We present this evaluation operator in the following subsection.

#### 3.1 Evaluation

The evaluation operation makes the evaluation of Tree Graph Views feasible without any particular algebra. The evaluation operation is particularly interesting for optimization purposes since it can prove equivalence rules.

In the Tree Graph View model, we need to define the *eval* function for each type of the model. Each TGV operation works on sets of tree patterns (ie: *tgv - rtp* is a set of *TreePattern* without the *ReturnTreePattern*). All operations use tree patterns on set of XML trees  $\tau$  and produce a new set of XML trees.

A set of constraints is expressed by a  $\sigma$ . Thanks to *Constraint Hyperlinks*, constraints can be found in a *Return Tree Pattern*. Dependences between Tree Patterns are given by hyperlinks. Since we work with sets of tree patterns, disjointed sets of tree patterns (tree patterns of one set are not hyperlinked to any tree pattern of the other set) infer a set rule:  $eval(tp_1 \cup tp_2) = eval(tp_1) \cup eval(tp_2)$ .

Some operations on trees need to be defined. These operations express trees manipulation, we will use it in the *eval* function:

- $\sigma$ : This operation apply constraints, from *Constraint Hyperlinks* linked to a *Return Tree Pattern*, on  $\tau$ . These constraints can prune trees when values are not verified, or modify values.
- $\pi$ : This operation projects XML tree values from  $\tau$  using *Directional Hyperlinks* as projections ;
- $\beta$ : This operation builds a new set of XML trees using *Return Tree Pattern* for tags hierarchy, and results from  $\pi$  for values ;
- $\alpha$ : This operation works on whole XML trees  $\tau$  to produce an aggregate result set using aggregate function ;
- $\phi$ : This operation restricts XML trees from  $\tau$  by filtering with tree patterns. Trees which does not contain nodes and node links association are pruned, others are kept with only nodes (and values) corresponding to the tree pattern.

Line	Functions	Precisions
1	$eval(tgv, \tau) = eval(rtp, \sigma_r(\tau_r))$	Decomposition on rtp and other evaluations with: $\tau_r = eval(tgv - rtp, \tau)$
2	$eval(rtp, \tau) = \beta(\pi(\tau))$	Values projection and tree construction
3	$eval(tgv - rtp, \tau) = eval(E_1 \cup \dots \cup E_n, \tau)$ $= eval(E_1, \tau) \cup \dots \cup eval(E_n, \tau)$	$E_x = stp_x \cup itp_x \cup atp_x$ $E_x$ are disjointed tree patterns sets
4	$eval(E_x, \tau) = eval(atp_x, \sigma_x(\tau_x))$	Decomposition on atp and others evaluations with: $\tau_x = eval(E_x - atp_x, \tau)$
5	$eval(atp, \tau) = eval(rtp, \alpha(\tau))$	$atp$ inherits from ReturnTreePattern with an aggregate function $\alpha$ on $\tau$ .
6	$eval(E - atp, \tau) = eval(E' \cup stp \cup itp_s, \tau)$ $= eval(E', \tau) \cup eval(stp, \tau) \cup eval(itp_s, \tau)$	$stp$ : Set of independent STP $itp_s$ : Set of ITP linked with a STP
7	$eval(stp, \tau) = \phi(\tau)$	Trees selection.
8	$eval(itp_s, \tau) = eval(itp, eval(tp_s, \tau))$	Itp and tp Decomposition.
9	$eval(itp, \tau) = \phi(\pi(\tau))$	Sub-trees selection.

Table 2: Tree Graph Views evaluation

Some manipulation operations on trees need to be defined, they will be used in the *eval* function. We will describe each of these operations (summarized in table 2)

**Line 1 Evaluation of a Return Tree Pattern.**

This evaluation builds an XML document from a set of trees formed (or built from) by other tree patterns. The *eval* function begins with the evaluation of a *rtp* on a set of trees  $\tau$ . This evaluation is equivalent to the evaluation of *rtp* with the restriction  $\sigma_r$  on  $\tau_r$ .  $\sigma_r$  is a set of constraints given by *constraint hyperlinks* linked to *rtp*. Values from trees that do not verify constraints are pruned.  $\tau_r$  is produced by evaluation of the remainder tree patterns:  $eval(tgv - rtp, \tau)$ .

**Line 2 Values Projection.**

An *rtp* builds an XML document with a set of values that are projections ( $\pi$ ) of  $\tau$ , these values are put in a tree given by a set of nodes in the *rtp* ( $\beta$ ). Relevant values of  $\tau$  are given by *DirectionalHyperlinks*  $\pi$  associated to the *rtp*. Thus, we obtain an XML result set from the union of a projection and a construction from *rtp* and  $\tau$ .

**Line 3 Evaluation of a tgv without the rtp.**

It is equivalent to the evaluation of a set of disjointed tree patterns  $E_x$ . These disjointed tree patterns are a set of *stp* and *itp* projected on an *atp*, the *atp* projects itself only on the

*rtp*. Since the evaluation function produces a set of trees, we can infer that the evaluation of a set of disjointed tree patterns is equivalent to the union of the evaluation of different sets of tree patterns:  $eval(E_1, \tau) \cup \dots \cup eval(E_n, \tau)$ .

**Line 4 Decomposition on atp.**

Since an *atp* inherits from *rtps*, we can consider it like a *rtp*. The evaluation of  $E_x$  is equivalent to  $eval(atp_x, \sigma_x(\tau_x))$  with  $\sigma_x$ , the set of constraints linked to the *atp\_x* and with  $\tau_x$ , the evaluation of  $E_x - atp_x$ .

**Line 5 Evaluation of atp.**

The evaluation of the *atp* is equivalent to a subset of  $E_x$  with tree patterns of  $E_x - atp$ . We just can have a difference with the *Return Tree Pattern* since an *Aggregate Tree Pattern* can have an aggregate function  $\alpha$ <sup>1</sup>, then the result set is obtained by the application of the function ( $\alpha$ ). However, we can have a difference with the *Return Tree Pattern* in the case the *Aggregate Tree Pattern* have an aggregate function  $\alpha$  since it is applied to obtain the result set.

**Line 6 Evaluation of disjointed tps.**

This line deals with the subset  $E - atp$ , which is a set of disjointed *stp*, *itp* and other sets of *atps* like in line 3. We found the equivalent evaluation:  $eval(E', \tau) \cup eval(stp, \tau) \cup eval(itp_s, \tau)$ , with *stp* disjointed with any *itps* and *atps*, *itp\_s* disjointed with *atps* and  $E'$  a set of tree patterns

<sup>1</sup>When an Aggregate Tree Pattern correspond to a nested query,  $\alpha$  is the *Identity* function.

with an *atp* which will be treated recursively on expression at line 3.

Line 7 **Evaluation of *stp***. The evaluation of an *stp* is a selection of a set XML trees from given collections. This trees are filtered by *nodes* and *nodelinks* association given by the Source Tree Pattern. We obtain a subset  $\tau$  of these XML trees for which tags correspond only to Tree Pattern's nodes.

Line 8 **Decomposition of an *itp* associated to a *tp***. The evaluation of an Intermediate Tree Pattern *itp<sub>s</sub>* associated to a Tree Pattern *tp<sub>s</sub>* is an evaluation of *itp* on a set of XML trees resulting from evaluation of *tp<sub>s</sub>*. *tp<sub>s</sub>* can be a Source Tree Pattern or an Aggregate Tree Pattern, then *itp* works on XML trees resulting of the evaluation of *stp<sub>s</sub>* or *atp<sub>s</sub>*. Else, *tp<sub>s</sub>* can be an other Intermediate Tree Pattern (*itp<sub>s</sub>*), we apply recursively operation.

Line 9 **Evaluation of *itp***. Finally, the evaluation of an *itp* is a node specialization which must verify the function  $\phi$  with *nodes* and *nodelinks* association from its root node given by the *stp* (line 8). Then, we obtain a subset of trees by the projection  $\phi$  on the selection  $\pi$  on  $\tau$ .

To conclude, we have defined an evaluation function  $eval(tgv, \tau)$  which provides the process to evaluates our model of XQueries translation. Our model is a logical evaluation of XQuery **which** is independent of any physical algebra. In addition, this model provides a logical optimizer based on equivalence rules that modifies *Tree Graph Views*.

### 3.2 Equivalence rules

An optimizer optimizes the query evaluation by rewriting the query using equivalence rules.

*Definition 1* introduces equivalence between two Tree Graph Views. Two tree graph views are equivalent if their evaluation are identical.

#### Definition 1 : equivalence

Two Tree Graph Views *tgv<sub>1</sub>* and *tgv<sub>2</sub>* are equivalent if they have the same evaluation on the same set of trees  $\tau$ :

$$eval(tgv_1, \tau) = eval(tgv_2, \tau)$$

An equivalence rules transform a Tree Graph View into another that must produces the same results set.

*Definition 2* introduces equivalence rules with the transformation  $\varphi$  that modifies the *tgv*. If the evaluation of  $\varphi(tgv)$  is equivalent to the evaluation of *tgv*, then  $\varphi$  is an equivalence rule.

#### Definition 2 : equivalence rules

An equivalence rule  $\varphi$  is a transformation applied to a Tree Graph View *tgv* that do not modify its evaluation:

$$eval(\varphi(tgv), \tau) = eval(tgv, \tau)$$

To prove the equivalence of a rule, we study the evaluation function *eval*. Since transformations are applied on functions of the evaluation, we can course the different stages of the evaluation (see in table 2) to see which level is targeted by transformations. Thus, we propose a simple algorithm that verify equivalence of a transformation by verifying each evaluation level.

As we can see in table ??, an algorithm that for each stage of Tree Graph Views evaluation, containing a set of tree patterns *E*, verifies equivalence (line 2) of each targeted operation *o* (line 1). This function is recursive on each decomposition of a stage (line 5). To describe transformations on operations, we need to detail different types of targeted operations :

- $\sigma$ : transformation on sets of Constraint Hyperlinks must respect selectivity of pruned trees ;
- $\pi$ : transformation on projections must comply value projection ;
- $\beta$ : transformation of document creation must comply schemas ;
- $\alpha$ : transformation on aggregates function must comply results ;
- $\phi$ : transformation on tree patterns must comply nodes restriction on result set.

Thus, equivalence between two Tree Graph Views obtained by a transformation is verified by modified operations. This operations define sets of trees that are evaluated, then transformations must respect these trees, not the manner to obtain it. This transformations are called equivalence rules if they do not modify set of trees, that is to say the evaluation. Moreover, since equivalence rules are applied under conditions, we can verify if a set of trees is respected by transformations while this conditions are true.

Equivalent Tree Graph Views are obtained by applying equivalence rules on TGV, but we now need to define a optimization strategy to orientate generation of Tree Graph views. This strategy will use these equivalence rules to define a way of optimization.

### 3.3 Extensible Optimization

Once we provide a set of equivalence rules into our optimizer, the optimizer defines a generation strategy to direct Tree Graph Views modifications. An extensible optimizer defines a set of equivalence rules to allow different evaluations selected by a research strategy. Some were proposed in Exodus [8], Starbust [32], EPOQ [5], and Volcano [21].

Like the Exodus system, we proposed to value transformation rules in order to direct rules choices. Most valued equivalence rules are set to an improvement coefficient between 0 and 1 on the Tree Graph View. Our optimizer selects THE more efficient rules with the biggest value in order to direct the optimization.

Unlike Exodus choice of valuation, based on history checks, our rule valuation consider the cost model valuation detailed in section 4. A theoretical improvement coefficient can be set by the cost model. Since equivalence rules change Tree Graph Views, we can evaluate two models and identify a theoretical coefficient ( $cost(tgv_1)/cost(tgv_2)$ ), that is very simple to implement.

The cost model can bring a difference between theory and effective evaluation of queries. However, since mediation architectures bring heterogeneity in source evaluation, we can't rely on historic evaluation. Indeed, the historic evaluation could be too variable between different equivalence rule execution. So, we rely on a directed mediation cost model which can approximate more effectively the cost of the evaluation, and consequently, the improvement coefficients.

## 4 Annotating the TGV

Equivalent TGVs can be generated from an initial TGV by using equivalence rules introduced in the previous section. Although results of their respec-

tive executions will be the same ones, the process of their execution will be different and will have different costs (in term of times, prices, connections, etc.) The classical solution for choosing the best execution plan is to compare plan costs using a cost model.

A TGV represents a global view of a query execution in the mediation system, so its different components can represent data located on several places (local mediator, distant sources).

We take advantage of the TGV global representation by annotating each or a set of TGV components with additional information as well for the costs as for the localization.

We propose a cost model somehow inspired by DISCO [26]. The mediator has a generic cost model. Each wrapper can export specific statistics and formulas to the mediator. The generic cost model is generally used with the exported statistics, but specific formulas exported by a wrapper can override generic formula. This approach gives a framework to compute the global cost of a query plan integrating local information on sources.

### 4.1 Cost model in a distributed heterogeneous semi-structured environment

The goal of a cost model is to estimate the cost of an execution plan for a query before its evaluation. A cost model for a data source may contain statistics and cost formula. Statistics describe characters of the data source, including collection properties and system information which is independent of data. Cost formulas are dependent on statistics and describe operations cost processed on the data source.

Elementary operations of an execution plan are distributed on the mediator and on the wrappers (sources). If the mediator knows all the statistics of sources, it can estimate the cost of these operations by classical cost model. On the other hand, if data sources do not reveal necessary cost information for the reason of autonomy, the mediator must use specific methods to estimate approximately the cost of an (or a set of) operation (s) processed by sources. State of the art on cost models is summarized in the table 4 that classifies the main cost model methods by data type.

<b>Data Type</b>	<b>Cost Model</b>			
<b>Relational Database</b>	Operation [27] [20] [38] [13] [24]	Calibration [16]	Sampling [36]	History [3]
			Adaptive Sampling [35]	Wrapper [23] [28]
<b>Object Oriented Database</b>	Operation [10] [6] [15]	Path [18]	Calibration [19]	Flora [17] [22]
<b>Semi-structured Data</b>	Operation [2]	Native SS [25]		XQuery [34]

Table 4: Cost models classifications

In table 4, methods with the name "operation" refer to the cost models for the elementary operations for each type of database. Generally, cost information such as sources statistics is necessary for these cost models. Each of the other methods has a specific name which indicates its main idea to calculate costs. Some methods need more source information than others, for example, the method by "Calibration" [16] [19] which estimates the coefficients of a cost model, need to know the access methods used by the source while this information is not necessary for the method by "History" [3], in which cost estimation of a new query is based on the history of queries processed.

## 4.2 Annotating TGV

Each TGV represents a global view of query execution in the mediation system. The optimiser can generate several equivalent TGVs for the same query according to the equivalence rules. All of these equivalent TGVs form the research space for the query. We should preview the cost of each TGV candidate in order to choose the TGV with the optimal cost to evaluate the query. A cost model is necessary for this cost estimation procedure.

We annotate each component (or a set of components) of TGV by a generic cost communication language (Section ). For an elementary operation, we annotate its cost formula; for a source, we annotate its statistics and treatment capabilities. All this information is collected together to calculate the whole cost of TGV.

The optimizer should distinguish the operations processed on the mediator and those processed on sources. Considering the limit treatment capabilities of sources, some operations can not be pro-

cessed by sources., so they should be processed on the mediator. As our common data model for the mediator is semi-structured, we will use the cost model in [34] to estimate the cost of these operations processed on the mediator.

On the other side, for the cost estimation of operations processed on sources, we can use the corresponding method in the table 4 for each data type. Sometimes, we can only estimate cost of a set of operations, due to the lack of necessary cost information. But this reduces the cost estimation charge and makes sometimes the cost model more efficient.

## 4.3 Generic cost communication language

We define a language to express the cost information in a uniform, complete and generic manner. This language should consider every cost model type and should allow wrappers' developers to export their cost information. In fact, this language completes the interface between the mediator and the wrapper (Figure 8). In our context, this language should be generic enough to express cost information of different parts of a TGV. It should also be capable to express cost for different optimization goals, for example, response time, price, energy consumption, etc.

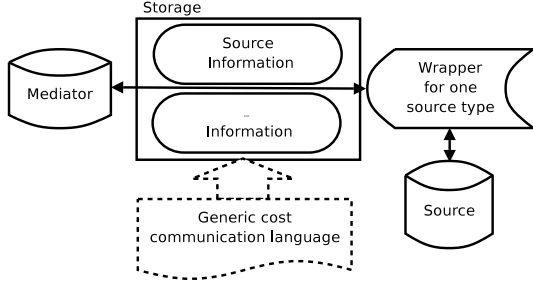


Figure 8: Role of generic cost communication language

In our system, the cost information communication is done in equation forms set. Each equation corresponds to a cost function which may be

defined by the source or by the mediator. Each component of TGV is annotated with an equation set in which the number of equations is unknown in advance. One function in a set may use variables defined in other sets.

The language should define some rules to ensure cost model's consistency. First, every variable should have somewhere a definition. Second, for reasons of being generic, there are no predefined variables' names. For example, in the language's grammar, we do not use a name "time" for a cost variable because the cost unit may be a price unit. It's the user of the language who gives the specific significant names to variables.

Cost formula	Cost language (using MathML)
$\left. \begin{aligned} Cost_{S1} &= Cost_{Restriction} + Cost_{Projection} \quad (1) \\ Cost_{S2} &= Cost_{Join} \quad (2) \\ Cost_{Total} &= Cost_{S1} + 2 * Cost_{S2} \quad (3) \end{aligned} \right\}$	<pre> &lt;cost source="mediator"&gt;   &lt;apply&gt;&lt;eq/&gt;&lt;apply&gt;&lt;ci&gt;CostS1&lt;/ci&gt;&lt;/apply&gt;   &lt;apply&gt;&lt;plus/&gt;     &lt;ci&gt;CostRestriction&lt;/ci&gt;&lt;ci&gt;CostProjection&lt;/ci&gt;   &lt;/apply&gt;&lt;/apply&gt;    &lt;apply&gt;&lt;eq/&gt;&lt;apply&gt;&lt;ci&gt;CostS2&lt;/ci&gt;&lt;/apply&gt;   &lt;apply&gt;&lt;ci&gt;CostJoin&lt;/ci&gt;&lt;/apply&gt;&lt;/apply&gt;    &lt;apply&gt;&lt;eq/&gt;&lt;apply&gt;&lt;ci&gt;CostTotal&lt;/ci&gt;&lt;/apply&gt;   &lt;apply&gt;&lt;plus/&gt;&lt;ci&gt;CostS1&lt;/ci&gt;&lt;apply&gt;&lt;times/&gt;     &lt;cn&gt;2&lt;/cn&gt;&lt;ci&gt;CostS2&lt;/ci&gt;&lt;/apply&gt;&lt;/apply&gt;&lt;/apply&gt; &lt;/cost&gt; </pre>

Table 5: Example of cost language

The Table 5 gives a simple example of our cost language. In this example, the query will be divided into several operations, processed on Source1(S1) and Source2(S2). The execution cost on S1 can be calculated by (1) and the execution cost on S2 by (2). The total execution cost of the query can be estimated by (3). There's a coefficient "2" for " $Cost_{S2}$ " because execution on S2 is more expensive than S1.

Our language is based on MathML [30], which allows us to define all mathematical functions in XML form. MathML is fit for cost communication between the mediator and wrappers due to our semi-structured environment. We use the *Content Markup* in MathML to provide explicit encoding for the cost formula. We'll just probably add some functions to MathML to define the grammar of our generic cost communication language. In table 5, we give the MathML description for each cost for-

mula.

Finally, to obtain an optimal TGV according to the optimization goal, we should just minimize or maximize a variable which is a parameter in a cost function. In the above example, we need to minimize the variable " $Cost_{Total}$ ".

## 5 Conclusion

XQuery is an XML querying language that provides a rich expressiveness. However, with this rich expressiveness, it is hard to provide an internal representation that can make the query efficiently processed. In this paper, we describe the TGV model, a model that we have conceived and we propose a method to rewrite XQuery expressions to make their evaluation efficient.

This method use the following steps:

1. Reduce the XQuery expression to a canonical form of XQuery using transformation rules
2. Model the canonized XQuery expression into the TGV model. The TGV model integrates the whole XQuery fonctionnalités and is an intuitive representation that supports annotations. Our optimizer is extensible and let the users define their own equivalence rules to transform TGV.
3. Prepare the evaluation phase by annotating the TGV. The annotation can be used in two cases. First, sources and local process location can be annotated on the TGV model in a mediator/wrappers architecture in a heterogeneous distributed systems. Second, cost model for an element of the XQuery expression, or for a subpart of the element can be annotated on the TGV. This can be done thanks to our generic, hierarchical cost model definition that makes the cost model of source type and cost type possible.

The whole XQuery evaluation process have been implemented in the XLive [12] mediator system. All XQuery expressions of the W3C use-cases [14] NS, PARTS, R, SEQ, SGML, STRING, TREE, XMP categories are evaluated correctly by our system, using TGV. For the moment, our system does not support the STRONG use-case category because of typing consideration. But we are planning to support typing in our system, so this last use-case category would be also implemented in our system.

With the TGV model, the XLive system<sup>2</sup> is now one of the heterogeneous integration system that support the most part of XQuery and is the only one to support extensible optimisation with cost support and annotation.

## Acknowledgment

The XLive is supported by the ACI Semweb project. The cost model is also supported by the ANR PADAWAN project.

<sup>2</sup>Downloadable on <http://www.prism.uvsq.fr/~travers/xlive>

## References

- [1] S. Abiteboul. Querying Semistructured Data. In *Proceeding of the 6th International Conference on Database Theory*, Delphi, Greece, 1997.
- [2] A. Abounaga, A. Alameldeen, and J. Naughton. Estimating the Selectivity of XML Path Expressions for Internet Scale Applications. In *The VLDB Journal*, pages 591–600, 2001.
- [3] S. Adali, K. Candan, and Y. Papakonstantinou. Query Caching and Optimization in Distributed Mediator Systems. In *ACM SIGMOD Int'l Conf. on Management of Data*, pages 137–148, Montreal, Canada, 1996.
- [4] S. Amer-Yahia, S. Cho, L. V. S. Lakshmanan, and D. Srivastava. Minimization of Tree Pattern Queries. In *SIGMOD Conference*, 2001.
- [5] S. Bjørnstad. The framework for EPOQ - an Extensible Object-Oriented Query Optimizer.
- [6] J. A. Blakeley, W. J. McKenna, and G. Graefe. Experiences Building the Open OODB Query Optimizer. In *ACM SIGMOD Int'l Conf. on Management of Data*, pages 287–296, Washington, D. C., 1993.
- [7] T. Bray, J. Paoli, and C. Sperberg-MacQueen. Extensible Markup Language (XML) 1.0 (W3C Recommendation), 1998.
- [8] M. J. Carey, D. J. DeWitt, G. Graefe, D. M. Haight, J. E. Richardson, D. T. Schuh, E. J. Shekita, and S. Vandenberg. The EXODUS Extensible DBMS Project: An Overview. In D. Maier and S. Zdonik, editor, *Readings on Object-Oriented Database Sys.* Morgan Kaufmann, San Mateo, CA, 1990.
- [9] Z. Chen, H. Jagadish, L. V. Laksmanan, and S. Paparizos. From Tree Patterns to Generalized Tree Patterns: On efficient Evaluation of XQuery. In *Very Large Data Bases*, pages 237–248, Germany, 2003.
- [10] S. Cluet and C. Delobel. A General Framework for the Optimization of Object-Oriented Queries. In *ACM SIGMOD Int'l Conf. on Management of Data*, pages 383–392, San Diego, California, 1992.

- [11] T.-T. Dang-Ngoc and G. Gardarin. Federating Heterogeneous Data Sources with XML. In *Proc. of IASTED IKS Conf.*, 2003.
- [12] T.-T. Dang-Ngoc, C. Jamard, and N. Travers. XLive: An XML Light Integration Virtual Engine. In *Proc. of BDA*, 2005.
- [13] D. Daniels and al. An Introduction to Distributed Query Compilation in R\*. In *2nd Int'l Symposium on Distributed Data Bases*, pages 291–309, Berlin, Germany, 1982.
- [14] D. Chamberlin, P. Fankhauser, D. Florescu, M. Marchiori, and J. Robie. XML Query Use Cases, september 2005. W3C. <http://www.w3.org/TR/xquery-use-cases>.
- [15] A. Dogac, C. Ozkan, B. Arpinar, T. Okay, and C. Evrendilek. *Advances in Object-Oriented Database Systems*, chapter METU Object-Oriented DBMS. Springer-Verlag, 1994.
- [16] W. Du, R. Krishnamurthy, and M.-C. Shan. Query Optimization in a Heterogeneous DBMS. In *Proc. of the 18th Int'l Conf. on Very Large Data Bases (VLDB)*, pages 277–291, Vancouver, Canada, 1992.
- [17] D. Florescu. *Espace de Recherche pour l'Optimisation de Requêtes Objet*. PhD thesis, University of Paris IV, 1996.
- [18] G. Gardarin, J.-R. Gruser, and Z.-H. Tang. Cost-based Selection of Path Expression Algorithms in Object-Oriented Databases. In *22nd Int'l Conf. on Very Large Data Bases (VLDB)*, pages 390–401, Mumbai (Bombay), India, 1996.
- [19] G. Gardarin, F. Sha, and Z.-H. Tang. Calibrating the Query Optimizer Cost Model of IRO-DB. In *Proc. of the 22nd Int'l Conf. on Very Large Data Bases (VLDB)*, pages 378–389, Mumbai (Bombay), India, 1996.
- [20] D. Gardy and C. Puech. On the Effects of Join Operations on Relation Sizes. *ACM Transactions on Database Systems (TODS)*, 14(4):574–603, 1989.
- [21] G. Graefe and W. J. McKenna. The Volcano Optimizer Generator: Extensibility and Efficient Search. In *ICDE*, pages 209–218, 1993.
- [22] J.-R. Gruser. *Modèle de Coût pour l'Optimisation de Requêtes Objet*. PhD thesis, University of Paris IV, 1996.
- [23] L. M. Haas, D. Kossmann, E. L. Wimmers, and J. Yang. Optimization Queries Across Diverse Data Sources. In *23rd Int'l Conf. on Very Large Data Bases (VLDB)*, pages 276–285, Athens, Greece, 1997.
- [24] L. F. Mackert and G. M. Lohman. R\* Optimizer Validation and Performance Evaluation for Local Queries. In *C. Zaniolo, editor, ACM SIGMOD Int'l Conf. On Management of Data*, pages 84–95, Washington, D. C., 1986.
- [25] J. McHugh and J. Widom. Query Optimization for Semistructured Data. Technical report, Stanford University Database Group, February, 1999.
- [26] H. Naacke, G. Gardarin, and A. Tomasic. Leveraging Mediator Cost Models with Heterogeneous Data Sources. In *ICDE*, pages 351–360, 1998.
- [27] G. Piatetsky-Shapiro and C. Connell. Accurate Estimation of the Number of Tuples Satisfying a Condition. In *ACM SIGMOD Int'l Conf. on Management of Data*, pages 256–276, Boston, Massachusetts, 1984.
- [28] M.-T. Roth, F. Ozcan, and L. Haas. Cost Models DO Matter: Providing Cost Information for Diverse Data Sources in a Federated System. In *25th Int'l Conf. on Very Large Data Bases (VLDB)*, pages 599–610, Edinburgh, Scotland, 1999.
- [29] N. Travers. *Optimization Extensible dans un Médiateur de Données XML*. PhD thesis, University of Versailles, 2006 (to appear).
- [30] W3C. Mathematical Markup Language (Mathml TM) Version 2.0 (Second Edition), 2003.
- [31] W3C. An XML Query Language (XQuery 1.0), 2005.
- [32] J. Widom. The Starburst Active Database Rule System. *Knowledge and Data Engineering*, 8(4):583–595, 1996.
- [33] G. Wiederhold. Mediators in the Architecture of Future Information Systems. *Computer*, 25(3):38–49, Mar. 1992.
- [34] N. Zhang, P. J. Haas, V. Josifovski, and C. Z. G. M. Lohman. Statistical Learning Techniques for Costing XML Queries. In *the 31st Int'l Conf. on Very Large Data Bases (VLDB)*, pages 289–300, Trondheim, Norway, 2005.

- [35] Q. Zhu. *Estimating Local Cost Parameters for Global Query Optimization in a Multidatabase System*. PhD thesis, University of Waterloo, 1995.
- [36] Q. Zhu and P.-A. Larson. Solving Local Cost Estimation Problem for Global Query Optimization in Multidatabase Systems. *Distributed and Parallel Databases*, 6::373–421, 1998.
- [37] M. Zloof. QBE : Query By Example, 1977.
- [38] M. T. Özsu and P. Valduriez. *Principles of Distributed Database Systems*. Prentice Hall, 2nd edition edition, 1999.