Tree Graph Views: On Efficient Evaluation of XQuery in an XML Mediator

Tuyêt-Trâm Dang-Ngoc Georges Gardarin Nicolas Travers
PRiSM Laboratory - University of Versailles
45, avenue des Etats-Unis
78035 Versailles CEDEX. France
tel: 33 (0)1 39 25 43 15
{firstname.lastname}@prism.uvsq.fr

ABSTRACT

XQuery is the emerging standard for querying XML data sources. XLive is a light XML/XQuery mediator developed at University of Versailles whose engine processes an XML algebra derived from the relational one extended to process in dataflow XML trees. The query optimizer translates a subset of XQuery in this algebra. To extend the optimizer's coverage of XQuery and better optimize query plans, we propose a representation of queries as graphs of trees, more precisely as tree pattern graphs interconnected by hyperlinks. Our structure called Tree Graph View (TGV) is an extension of the Generalized Tree Pattern graph proposed in [5] as a concise and practical representation of an XQuery request. It is designed to be a more intuitive model of queries and to allow direct optimization before generating the physical execution plan. TGV lends itself to simple algorithms to generate efficient algebraic execution plans. Moreover, it is effective for view translation and query simplification, and for taking into account source capabilities. We are currently implementing it to support the new XLive optimizer.

Keywords

XQuery evaluation, Mediation, Tree pattern, XML algebra, Views, Optimization, Execution plan.

1. INTRODUCTION

In this paper, we address the problem of query optimization in an XML mediator supporting XQuery. An increasing number of XQuery-based information integration platforms are available. Query optimization in a mediator is a difficult task, which has been addressed in a few papers [12] [14]. Several graph-based models have been introduced for centralized XQuery processing, including Tree Pattern and Generalized Tree Pattern [5]. The adaptation of such models to distributed query processing remains to be done. We propose an extended representation of XQuery called the TGV (Tree Graph View), which is convenient for query simplification, optimization, and transformation in a mediator.

At the University of Versailles, we have designed and implemented two versions of a mediator [7]. The first has been commercialized and is now distributed in open source (see www.xquark.org). The latest is a research vehicle named XLive. XLive is an XML mediator, dealing with a significant subset of XQuery. The architecture is able to integrate and query relational or XML sources in XQuery. Wrappers can be accessed through Web services or directly through a Java API. Wrappers translate a subset of the common query language (XQuery) into the source native query language and map the source native result format into the common format (XML). Wrappers can have different capabilities, i.e., do simple selection, navigation with XPath query, perform FLWR with joins, etc. The mediator decomposes global XQueries in local ones plus integration operations (union, join, reconstruction) handled by the mediator.

One main objective of the mediator is to generate from an XQuery request an optimized execution plan. The plan is an XML algebraic operation tree ready to be evaluated, the leaves by wrappers, and the rest by the mediator. The construction of an execution plan tree directly from an XQuery request is very complex although equivalence rules for nested queries and reconstruction have been defined [12] [14]. Moreover, execution plans are not convenient for all optimizations. Starting from [5], we design a convenient model for representing and optimizing XQuery. An XQuery is modeled as a graph interconnecting annotated tree patterns called a TGV. This representation can be seen as an extension of relational query graphs to XML trees in place of relations. Similar representations have been proposed (XQBE [4], MIX [3]), but they are applied to the design of user-friendly interfaces and not to query processing.

The XLive query runtime is dataflow-oriented and built around an extended relational algebra for XML, known as the XAlgebra. Several algebras have been proposed for XML processing, among them tree algebras manipulating set of trees [9] [11] and extended relational algebra manipulating non-1NF relations [6]. Our algebra is hybrid in the sense that non-1NF relations are used to store pointers to tree nodes. The original idea of our algebra is to represent XML documents as tuples of references to virtual DOM trees, called XTuples. The mediator evaluates XTuples all along our query plans composed of XAlgebra expressions and constructs the

XQuery result. To create the algebraic plan, rules for transforming the query TGV into XAlgebra operators are enumerated. Furthermore, the query TGV is suitable for various optimizations and operations as handling sources capabilities and applying view rewriting.

This paper is organized as follows. In the next section, we survey as background the GTP graph structure and describe the proposed TGV structure. In section 3, we present the XLive XML algebra and sketch the execution plan construction method. In section 4, we show how to optimize execution plans, handle source capabilities, and rewrite queries on views using XQuery TGVs. The conclusion summarizes our contribution and suggests some extensions.

2. XQUERY MODELIZATION

2.1 Generalized Tree Pattern (GTP)

A well-known structure for modeling XML queries is the so-called Tree Pattern Query (TPQ) [5]. A TPQ is a tree with nodes labeled by variables together with a formula specifying constraints on tags, attributes, and contents. The tree consists of two kinds of edges – parent-child (pc) and ancestor-descendant (ad) edges. The semantics of a TPQ is based on the notion of pattern match, a mapping from the TPQ nodes to the database nodes satisfying the formula.

A Generalized Tree Pattern [5] 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 Figure 1. Notice the dotted line, the join conditions, and the block levels (e.g., 1.1.0). Block levels identify links between TPQs and is defined by the level of hierarchy in the group number.

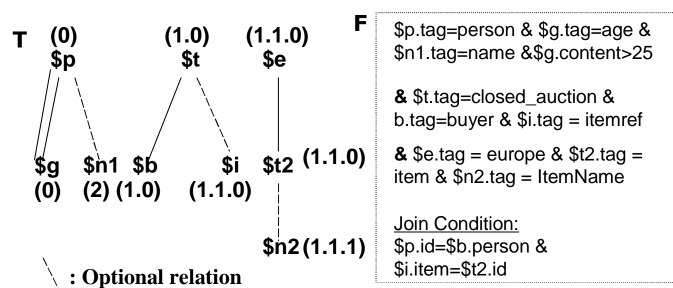

Figure 1. Generalized Tree Pattern

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 is not possible.
- Let and functions are not integrated.
- Tags, relations, and constraints are embedded in a Boolean formula difficult to read.

All in all, the GTP representation is not very intuitive, as dependencies between tree patterns are not explicated except in formulas. 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 are currently implementing in the XLive mediator for XQuery processing.

2.2 Tree Graph View (TGV)

As in GTP, we represent XQuery requests using tree patterns. A query is mapped into a bi-graph called a Tree Graph View (TGV). This is really a graphical view of a graph whose nodes are trees. A TGV is composed of a set of tree patterns $\mathcal{T}$ and a set of hyperlinks $\mathcal{H}$.

2.2.1 Tree pattern

A tree pattern similar to the TPQ is associated to each XML collection. As usual, a tree pattern is composed of nodes and edges. Annotations are added on nodes and edges. Nodes are labeled with a label representing a tag or a data source (root node), an optional selection predicate representing a filter applied to the node, and an optional function expression to apply to the node after filtering. Thus, a node **N** is annotated by a structure (*label*, *predicate, function expression*). Let $\mathcal{N}$ be the set of nodes of the tree pattern. An edge between two nodes represents a parent-child (simple line) or ancestor-descendant (double line) relationship. They can also be optional or mandatory as in TPQ. Thus an edge **E** linking two nodes is defined as **E**(*source_node*, *destination_node*, *descendant*, *mandatory, multiplicity*), where *source_node* and *destination_node* belong to $\mathcal{N}$, descendant and mandatory can be true or false, multiplicity means that these element should be nested within its parent (see below). For edges, we keep the representation of the GTP, with double line for ancestor-descendant link, solid line for mandatory branch, and dotted line for optional one.

We distinguish a special tree pattern – called the *return tree pattern (RTP)* – that models the return clause of a query. In general, nodes in the return tree pattern have no predicate attached; all edges represent parent-child relationships. In our graphical representation, common tree patterns are represented in bubbles while the return tree pattern is enclosed in a box.

Every tree pattern matches the schema of the represented collection, including the return tree pattern, which matches the result schema. When processing queries each XPath generates new labels into the tree pattern. An attribute is formalized as a label with a "@" in prefix. When an XPath is referenced only into a return clause, it is not mandatory in the XML result fragments as an empty tag can be generated. In this case, a dotted line is used to represent the XPath extraction, meaning that null values are allowed.

In the context of a mediator, we distinguish source tree patterns (STP) from other tree patterns. A STP models a query on a data source collection. It has to match with the schema of the data source. It is not derived from another tree pattern. The label of the root of the tree pattern is the source name. Figure 2 portrays a *source tree pattern* on "**source1**". A restriction is applied on the XPath (*source1//a/[b = 5]*). A count function is set on *source1//a/c*; a is linked by a dotted line to c as this path is not mandatory and is only useful for the result reconstruction; the "*0" on c means it should be nested inside its parent a (before counting). Finally, the XPath source1//a//d/@id is optional and useful for results as well.

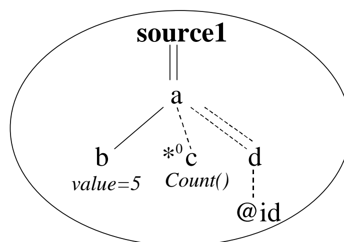

Figure 2. Source Tree pattern

Non-source and non-result tree patterns are called intermediate tree patterns (ITP). An ITP is resulting from a mapping from one or more other tree pattern. This tree pattern is useful to model queries including a variable created from another one (e.g., *for $i in $j*) or intermediate results coming from a filtered data source.

A problem when processing queries is to model nesting operations. To represent the fact that a node should be nested in a parent, we introduce an additional multiplicity mark graphically denoted by a "*". Nesting can be requested simultaneously on several child elements of a given node. To denote that, we use an index; thus, two stars on the out edges of the same parent with the same level of nesting (e.g., *1 and *1) mean nested together within the parent.

2.2.2 Hyperlinks

Tree patterns are connected together by hyperlinks. Hyperlinks connect two nodes belonging to different tree patterns. There can be several hyperlinks from one tree pattern to another. In general, hyperlinks are labeled by a comparison predicate chosen among {=,<,<=,>=, ≠}. An hyperlink models a correspondence between two nodes:

the instance node values satisfying the query shall satisfy the comparison predicate. As any edge, an hyperlink can be mandatory or optional (meaning null value possible). An hyperlink **H** belonging to the set $\mathcal{H}$ of all the hyperlinks of the TGV is represented by **H**(*source_node*, *destination_node*, *type*, *mandatory, multiplicity*), where *source_node* and *destination_node* belong to $\mathcal{N}$, *type* can be of type projection or of type join, *mandatory* can be true or false, and *multiplicity* is the nest index saw in the previous subsection.

The TGV is a declarative model that represents as a graph of trees the relevant fragments of the database for the query. However, to clarify the description and be complete, we introduce some procedural features in the model (that is partially the case for the multiplicity factor and the distinction between STP, ITP, and RTP). In this line, we distinguish join hyperlinks and projection hyperlinks. Joins hyperlinks connect two tree patterns that are not return tree patterns. It corresponds to a join condition between paths of two collections. It is in mandatory for full join and optional for outer-join (we should distinguish left and right outer-joins partly dotted). Figure 3 illustrates a join hyperlink between two sources (“source1” and “source2”) on different XPaths: *source1//a//d//@id = source2//a2/b2/text()*.

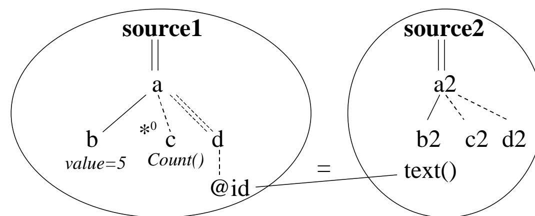

Figure 3. Join Hyperlink

Projection hyperlinks connect two tree patterns where the projected tree pattern results from a mapping of the other tree pattern. One of the tree patterns can be (and often is) a RTP. A projection hyperlink is always labeled with an equal predicate. It is optional in results if not constrained elsewhere by a predicate or full join.

The model is powerful enough to represent let clauses. A let clause generates a new tree pattern. This tree pattern is matched by a tree that represents the expression defining the let variable (in general an XPath). When this tree pattern is produced from an existing tree pattern, hyperlinks labeled by the equal predicate are created with the parent node. The result of a let clause is often nested under the root label by adding the multiplicity annotation (*i).

For example, the query “*for each restaurant, select its identifier and name, the number of proposed menus, and the list of employees with their name, age and phone*” is represented by the TGV of Figure 4.

```
for $r in collection(“source1”)//restaurant
let $m := $r//menu
```

```
where $r/category = "5"
return
  <restaurant id="{$r/@id}">
    {$r/name}
    <nb_menu>{count($m)}</nb_menu>
    <employees> {
        for $p in collection("source2")/person
        where $p/work/employer/text () = $r/@id
        return
         <employee>
           {$p/name}
           <comment>
               {$p/age}
               {$p/phone}
           </comment>
         </employee>}
    </employees>
</restaurant>
```

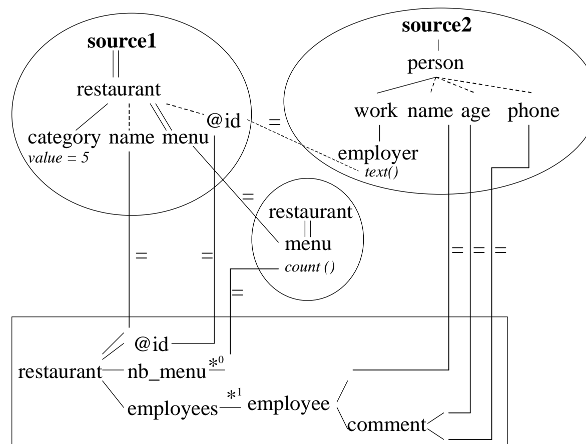

Figure 4. An example of TGV

In Figure 4, the two sources tree patterns are connected by a join hyperlink. The Let clause duplicate the sub-tree of root restaurant and descendant menu. *Source2* is a nested query, thus a nesting factor (*0) is added on the relevant element of the RTP.

We now give a more formal definition of the TGV model. A TGV is defined by a set of hyperlinks **H** and a set of tree pattern **T**. In the table of Figure 5, we characterize the components of the model using a grammar. The set of

tree patterns is partitioned in three subsets: STP, ITP and RTP, thus the TGV can be expressed as in line **(1)**. Before optimization, in most cases, each bubble (STP or ITP) just have one tree pattern inside. But we will see in section 4 that bubbles can be merged, thus there could be several tree patterns in one bubble linked by internal hyperlinks. Thence STP (line **2)** and ITP (line **3)** are defined as one or more tree patterns and a set of hyperlinks. An RTP is composed of one Tree-Pattern (line **4)**. Lines **(5)**, **(6)**, **(7)** and **(8)** describe the tree pattern components (edges and node) and hyperlinks. A Bubble Tree Pattern (BTP) is a generic term designing STP, ITP or a RTP (line **9)**.

	Name	**Grammar Definition**	**Description**
(1)	Tree Graph View	TGV := (TP*, H*) TGV := (STP*, ITP*, RTP, H*) TGV := (BTP*, H*)	Definition of TGV in terms of TPs and hyperlinks.
(2)	Source Tree Pattern	STP := (TP+, H*)	A source TP may include one or more TP.
(3)	Intermediate Tree Pattern	ITP := (TP+, H*)	
(4)	Return Tree Pattern	RTP := TP	A return TP is only one TP
(5)	Tree Pattern	TP := (N*, E*)	A TP is a graph.
(6)	Tree Node	N := (*string*, predicate_expression*, function_name*)	A node is composed of a node name, 0 or more predicate expression, and 0 or more function names
(7)	Tree Edge	E :=(N, N, *boolean*, *boolean*, integer)	An edge is composed of source_node ; destination_node ; descendant (// : true) or son (/ : false) ; mandatory (true) or optional (false) , multiplicity
(8)	Hyperlink	H := (N, N, *boolean*, t*ype* , integer)	source_node, destination_node, type : projection or join, mandatory (true) or optional (false), multiplicity
(9)	Bubble Tree Pattern	BTP : = STP | ITP | RTP = (TP+, H*)	A BTP is a generic bubble that may include several TPs.

Figure 5. Definition for TGV terms

3. EXECUTION PLAN

3.1 XML Algebra

The XLive runtime engine executes query plan expressed in XAlgebra [7]. XAlgebra includes both relational operations to process tables of references to nodes and navigation operations in the XML trees. The algebra is a physical algebra in the sense that algebraic expressions are used to process XML flows and that algorithms are directly implementing the operators.

3.1.1 XRelation and XTuple

We introduce *XRelations*, which can be considered as a special case of object relations, domains being XML trees. Classically, an XML tree is a set of labeled ordered rooted trees. In addition, cross-tree hyperlinks can be supported as special edges. With XRelation, domains are XML trees of given path set. Attributes are XPath expressions referencing nodes in the XML trees. Each attribute can be multi-valued, i.e., refers several sub-trees. XRelations are ordered collections of XTuples. Thus, each XTuple is composed of XPath named *XAttributes*, values of which reference sub-trees in the collection of trees. As a result, the schema of an XRelation is of type R(XPath+, *[Path+]*), where XPaths are defining the attributes and Path's compose the path set of the XML trees.

Figure 6 shows an example of an XRelation composed of two XTuples. The schema of the XRelation is:

```
R(person/fname, person/address, person/address/street, person, book/title, book/author/lname, book/date,
[person/fname,person/lname,person/address, person/address/street,person/address/town,book/title,
book/author,book/author/lname, book/date ] )
```

An XTuple refers to nodes and can be perceived as an index of XML trees.

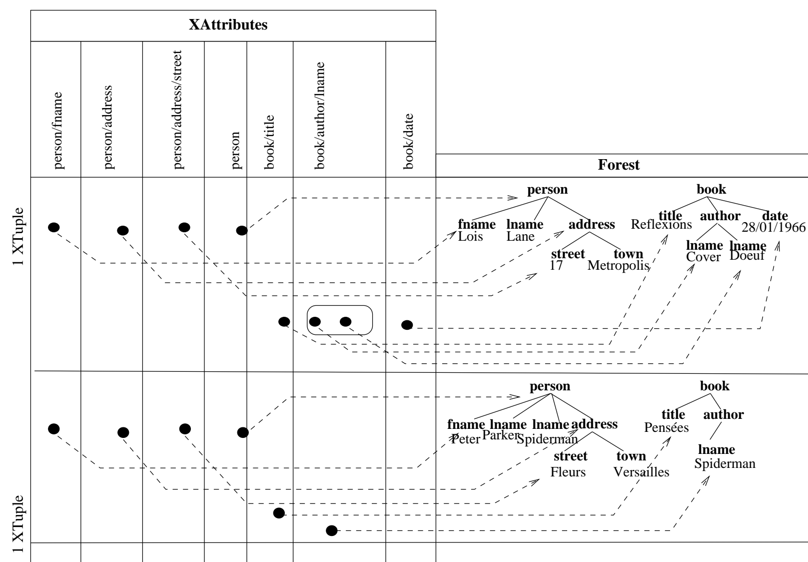

Figure 6. An Example of an XRelation

3.1.2 XAttribute annotation

We introduce annotations on XAttribute in order to distinguish attributes behavior in the request evaluation. By default, attributes are unnested and mandatory. We introduce the ‘?’ sign for specifying an optional attributes and the ’^{Nx}‘ sign to specify a nested attribute of nest group *x*.

With these annotations, the schema of an XRelation has the following form:

R ((XPath Annotation?)+ [Path+])

For example:

R1 (person/fname?, person/lname^{N1}, person/address^{N2}/street?, person/address^{N2}/town?, [person/fname,person/lname,person/address, person/address/street,person/address/town])

means that in the XRelation *R1*, the *fname* is optional, there is a mandatory *lname* and if there are several lname, it must be nested (*N1*) and the *address* node is also nested in another group (*N2*) if multi-valuated and have the optional connected nodes *street* and *town*.

3.1.3 XOperators

Except for XSource and XReconstruction, all operators of the XAlgebra receive one or more XRelation as input and return an XRelation as output. In general, XRelations are modified directly in memory. We give an overview of the main operators below.

XSource (source, xquery, xpaths*) → XRelation

XSource is the starting operation to process XML data sources. XSource takes the name of a source, an XQuery applying to this source, and the XPath describing the XAttributes in the result XRelation. XSource transforms an XML source in an XRelation of attributes a sequence of XPaths.

XRelation.XRestriction (*predicate*) → *XRelation*

The XRestriction operator filters XTuples satisfying conditions on attribute values.

XRelation.XProjection (*xpath**) → *XRelation*

The XProjection removes attributes that are not in the specified xpath sequence.

XRelation.XJoin (XRelation, predicate) → XRelation

The XJoin operator joins two XRelations on attribute values satisfying the predicate.

XRelation.XNest (XAttribute*) → XRelation

If there is an XAttribute argument, the XNest operator acts as a group-by operation, merging in the same XAttribute several references on different nodes. If there are no XAttribute, the XNest operator just merges all XTuples in one, merging all attributes of each column together. Note that for nesting, every attributes can be an ordered list of reference. And there can be as many levels of ordered lists on the references.

XRelation.XFunction (function, arguments*) → XRelation

Predefined function and external functions as min, max, count, and avg are applied using the XFunction operator.

```
XRelation.XConstruction (construction) → XML
```

The XConstruction operator extracts XML documents of given schema from the given XRelation.

With this algebra, query execution plans can be expressed. For example, for the example request of Figure 4, the following execution plan can be specified:

```
R1 := XSource("source1",
"for $r in collection("source1")//restaurant
 where $r/category = "5"
 return <results>{$r/name} {$r//menu} {$r/@id}</results>",
        {restaurant/name?, restaurant/menu?, restaurant/@id})
R2 := R1.XProjection("restaurant/menu")
R3 := R2.XNest ()
R4 := R3.XFunction ("count", "restaurant/menu")
R5 := XSource("source2",
"for $p in collection ("source2")/person
return <results>{$p/name} {$p/age} {$p/phone} {$p/work/employer/text()}</results>",
         {person/name?, person/age?, person/phone?, person/work/employer/text()})
R6 := R5.XNest () ;
R7 := R1.XJoin (R6, "person/work/employer/text()=restaurant/@id")
XConstruction (R7, "<restaurant
id="{$r/@id}">{restaurant/name}<nb_menu>{count(restaurant/menu)}</nb_menu>
<employees>{<employee>{person/name}<comment>{person/age}{person/phone}</comment></employee>}
</employees></restaurant>")
```

3.2 From TGV to Execution Plan

The TGV is the basis for query transformation and optimization. From a given TGV, a query plan can be generated. A query plan is valid if the relevant wrappers can execute the XSource operations that it contains. Generating valid query plan is not an easy task as sources have limited capabilities. The goal of the query optimizer is to generate the most efficient valid query plan, according to some cost model. We now detail how to generate a logical query plan from a TGV.

Each STP (source tree pattern) is transformed into one or more XSource operation(s) followed by a XUnion in case of multiple XSource operations. The tree pattern is rewritten into a mono-collection request and the required XAttributes are determined. The collection is possibly partitioned on multiple sources and thus an XSource operation must be generated for each relevant source. As the mediator maintains the path set describing each source (these metadata are acquired at connection time), relevant data sources for a STP are determined from the

metadata. XPath with ancestor-descendant relationships (double line) can also be expanded in fully documented XPath (with only parent-child relationships).

Basically, we cannot make any assumption on XML query capabilities of the sources. The only reasonable requirement on an XML source is that it can at least provide a *scan* method on a collection, i.e., return all the stored XML documents without any filtering. We will take into account sources capabilities and include the source filtering when possible in section 4.3.

Each Source Tree Pattern enclosed into a bubble can be basically transformed into a sequence of XOperator using the following steps:

1. The root of the Tree Pattern is transformed into an an XSource (*source_collection*, *request*, *attributes*) operator, where the source_collection is the name of the root, request is the scan request and attributes is '*', meaning that all attributes of the XML document are requested.
2. The whole set of conditioned nodes determines the relevant path set {$XPath_i$} of the Tree Pattern including leaves and internal nodes. They are used to determine the XAttributes of the XProjection ($XPath_0,\ldots,XPath_n$) operator. The resulting XAttributes are the paths of relevant nodes, considering the / (resp. //) notation while navigating in the Tree Pattern in place of the single (resp. double) edges. Note that nodes reached by a dotted line are represented by an XAttribute annotated by the optional sign '?' and nodes annotated by a '*x' sign are represented by an XAttribute annotated by the '^{Nx}' sign.
3. All the nodes of the Tree Pattern associated with a predicate are used to generate XRestriction operators conditioned by the predicate and returning the corresponding XAttributes.

A whole bubble is mapped to a linear execution subtree composed of a starting XSource operator, an XProjection and a sequence of zero, one or more XRestriction for each constrained XAttributes. This yields the following sequence of operations:

R1 := XSource (*source_name*, *scan*, **)*
R2 := R1.XProjection ($XPath_0$, ,$XPath_n$)
R3 := R2.XRestriction. ($p_1(XPath_0)$)
Ri = R(i-1).XRestriction. ($p_m(XPath_m)$)

Hyperlinks of type "joins" are transformed into the XJoin operator. From joins hyperlinks joining two XRelation $\mathbf{R_i}$ and $\mathbf{R_j}$, the operator (theta-join, equi-join, semi-join) on the joined XPaths from $\mathbf{R_i}$ and $\mathbf{R_j}$ is used to create the predicate $\mathbf{p_{ij}}$. Thus, XJoin operations are generated of the following form:

R_i..XJoin (R_j, p_{ij}) → R_k

Hyperlinks of type "projection" entering the RTP (result tree pattern) are transformed into the XReconstruction operator. The reconstruction template is simply expressed as an XML document template with the XAttributes of the input XRelation as values. The reconstruction template is passed as an argument to the XConstruction operator.

4. TGV-BASED OPTIMIZATION

4.1 TGV Transformation

A TGV is in a sense a declarative definition of the relevant data to process for computing the result of an XQuery. To define transformations, we use the following notations:

- *tgv* is a TGV with the set of tree patterns $\mathcal{T}$ and the set of hyperlinks $\mathcal{H}$.
- *btp* is a Bubble Tree Pattern; according to the definition, $btp = (tp_1, ...tp_n, h_1, .., h_n)$ with $tp_1, .., tp_n$ belonging to the set of the Tree Pattern $\mathcal{T}$ and $h_1, .., h_n$ being internal hyperlinks.
- H is the set of hyperlinks linked to and from *btp*; $H = H_s \cup H_d$ *with* $H_s = \{h \in \mathcal{H} / h.source \in (tp_1, ...tp_n)\}$ the set of outgoing hyperlinks and $H_d = \{ h \in \mathcal{H} / h.destination \in (tp_1, ...tp_n)\}$ the set of incoming hyperlinks.

To reduce data to process, to simplify a TGV, and more generally to perform logical optimization, several transformations can be applied to a TGV:

1. *Clone bubble*. A bubble *btp* can be duplicated, which requires adding the correct set of ingoing hyperlinks and the correct set of outgoing hyperlinks. This corresponds to adding an intermediate XRelation in the query plan. The Intermediate Tree Pattern *itp* is constructed by cloning the tree patterns $(tp_i \rightarrow tp'_i)$ and the internal hyperlinks $(h_i \rightarrow h'_i)$, with $itp = (tp'_1, ...tp'_n, h'_1, .., h'_n)$. The set of hyperlink $\mathcal{H}$ of the TGV must be modified. We must replace the set H_s of outgoing hyperlinks by a set of hyperlinks H_{s1} with the same source node but pointing to the image of the Tree Pattern in the Intermediate Tree Pattern and a set of hyperlinks H_{s2} from these point to the original destination. $H_{s1} = \{h_f / h \in H_s, h_f.source = h.source \text{ AND } h_f.destination = image (h.source)\}$ and $H_{s2} = \{h_f / h \in H_s, h_f.source = image (h.source) \text{ AND } h_f.dest = h.destination\}$. In the same way for the set of incoming hyperlinks, $H_d = \{h_f / h \in H_d, h_f.source = h.source \text{ AND } h_f.destination = image (h.source)\} \cup \{h_f / h \in H_d, h_f.source = image (h.source) \text{ AND } h_f.dest = h.destination\}$. We use clone bubbles to have an intermediate relation, for generating an equivalent execution plan, but also for taking into account wrappers capabilities as shown in section 4.3.
2. *Remove bubble*. A bubble can be removed either because all entering hyperlinks are optional joins or the incoming hyperlinks are similar to the outgoing ones. This corresponds to suppressing useless conditions or

useless intermediate XRelations. With the previous notation, a bubble *btp* -with sets of incoming hyperlink H_d and outgoing hyperlink H_s- can be removed if $\{ \forall h \in H_d ,/ (h.type = "join" \text{ AND } h.mandatory = false) \text{ OR } (\exists h' \in H_s / h.destination = h'.source)\}$. The TGV *tgv* would then become *tgv \ btp*.

3. *Migrate predicate*. When a node is constrained by a predicate, the predicate can be migrated to other nodes through equality hyperlinks (modeling equi-joins or projections). This is fairly similar to sideway information passing in deductive database. Predicate migration can be generalized with transitivity rules on theta-comparators (>, <, >=, <=). If $(h \in H_1 \cap H_2)$ *AND (h.type = "join") AND (h.predicate = "=")* then the linked nodes *h.source* and *h.destination* will have the following predicates: $(h.source.predicate \cup h.destination.predicate)$
4. *Migrate function*. When a function is applied to a node, the node value can be passed to another node through an equality link. Let *btp1*, *btp2* two BTP with respectively H1 and H2 the hyperlinks (incoming or outgoing) connected to them. If $(h \in H_1 \cap H_2)$ *AND (h.type = "join") AND (h.predicate = "=")* then the linked nodes *h.source* and *h.destination* will have the following predicates : $(h.source.function \cup h.destination.function)$
5. *Migrate nesting*. When a nesting (*i) is applied in a node, it can be migrated with the associated function if any through projection hyperlinks.
6. *Remove hyperlink*. An hyperlink can be removed if it computes nodes not mandatory in the final result nodes and in conditions. It can also be removed in case where the join predicate is asserted by other hyperlinks.
7. *Add hyperlink*. Hyperlinks between nodes that are set equals by other hyperlinks can be added. This corresponds to the well-known transitivity of join rule (a=b and b=c ➔ a=c). Let btp_1, btp_2, btp_3 three BTPs with respectively H_1, H_2 and H_3 the hyperlinks (incoming or outgoing) connected to them. If $(h_1 \in H_1 \cap H_2)$ *AND* $(h_2 \in H_2 \cap H_3)$ *AND* $(h_1.type = h_2.type = "join")$ *AND* $(h_1.predicate = h_2.predicate = "=")$ then we can add an hyperlink $h_3 \in H_1 \cap H_3$ with type join and edge labeled with equal.
8. *Propagate optional*. Certain nodes in RTP are optional, in general when no predicate apply to them. Other are mandatory. When a node is source of another, the optional and mandatory feature can be propagated with the rule optional and mandatory gives optional.
9. *Remove node*. A node can be removed if it has no predicate attached, nor any hyperlink.
10. *Merge bubbles* : For some optimization (see section 4.3, taking sources capabilities into account), bubbles can be merged. Their tree patterns and hyperlinks are then enclosed in the final merged bubble. Let $btp = (T,H)$ [resp. $btp' = (T', H')$] be a Bubble Tree Pattern of a TGV tgv $(\mathcal{B}, \mathcal{H})$, with $btp \in \mathcal{B}$ [resp. $btp' \in \mathcal{B}$], and T [resp. T'] be a set of tree patterns, IH [resp. IH'] be a set of internal hyperlinks, and H [resp. H'] $\in \mathcal{H}$ be the

set of hyperlinks connected to *btp* [resp. *btp'*]. The set of hyperlinks joining *btp* and *btp'* is then $H \cap H'$. The merged btp_m would then be defined by $(T \cup T', IH \cup IH' \cup (H \cap H'))$. Thus, the final *tgv* would have the following form : $(\mathcal{B} \setminus (btp \cup btp1) \cup btpm, \mathcal{H} \setminus (H \cap H'))$.

In summary, several transformations can be applied to a TGV in order to support query optimization. These transformations are in general applicable under certain conditions, but are useful for taking into account query optimization, source capability handling, and view query rewriting.

4.2 TGV Optimization

Classical logical optimization rules can be applied on the TGV, as graph transformations. For example, pushing selection to sources for reducing numbers of processed XTuples consists in migrating predicate to SPT, and pushing projection for reducing number of XAttributes of each XTuples consists in eliminating useless hyperlinks from the RPT. More advanced optimization can be done, as removing useless subqueries (remove bubble) and taking advantage of transitivity of equality and other comparison operators.

Using the metadata information, we can get more information for optimizing the TGV. For example completing path as seen above or identifying collections located on the same source is possible. When two collections are located on the same source, the source bubbles can be merged to generate only one XSource operator.

In our mediation context, sources and the mediator can have costs statistics and models. We plan to use a generic cost model as defined in DISCO [13]. Thus, a cost model to estimate plan cost could be used to drive the TGV transformations and plan generation.

4.3 Taking Advantage of Source Query Capabilities

Since the data sources (DBMS, Web server, search engine, etc.) may be very heterogeneous, their wrappers can have different capabilities for query processing. If a wrapper is not able to handle functionality an XOperation (restriction, joins, etc.), the mediator must handle it [15]. Thus, a valid TGV must be generated from the query basic TGV. For each STP, non-possible navigation, condition and function should be isolated (e.g., colorized). The most basic requirement that a source must be able to do is the scan operation. In this case, everything but the root of the tree pattern would be colorized.

Before generating the execution plan as described in 3.2, the plan generator analyzes the TGV and the colorized part of the TGV. A STP associated to a source that performs the minimal scan operation is transformed into the sequence XSource (source_name, scan, *)-XProjection-XRestriction

Dedicated XML data sources have operators improved for queries optimization, by using internal indexing and having direct disk access for data location, page loading, etc. So if such a source is able to do a particular XOperation, it is better to delegate the operation to the source instead of doing it on the mediator. The problem of handling query capabilities of an XML data source is then how to formulate the query passed to the XSource operator. For example, in most cases, XML databases know how to handle simple XQuery with selection and projection. Then in the best cases, for a Source Tree Pattern applied to $source_i$, the sequence XSource (*source_name*, *scan*, *)-XProjection ($XPath_0$, $,XPath_n$)-[XRestriction $_{p_1(XPath_0)]^*}$ can be merged into a single one XSource (*source_name*, *query*, *xpaths*) where *xpaths*=($XPath_0$, $,XPath_n$) and query is of the following form:

```
for $var in source_i
where $var/XPath_1 and $var/XPath_2 … and $var/XPath_n
return <result> {XPath_0} … {XPath_n)</result>
```

with *xpaths* in the return part and the set of predicates given in the XRestrictions in the where part.

Notice that joins are generated out of the XSource operations. If the source is able to handle the join operation and if the collections are on the same source, pushing join to the source has to be done at physical optimization directly on the query plan.

4.4 View Optimization

An XML view is a virtual collection defined by an XQuery [1]. The definition query can be represented as a TGV. A query on the view corresponds to another TGV applied to the RPT of the view TGV. The query modification process can be expressed as “merging TGVs”. The RPT of the view definition should be transformed into an IPT. Then, removing useless SPT or IPT, migrating predicates and functions towards the sources, etc., should be applied to optimize the global TGV.

To demonstrate the ability of TGVs to optimize queries on views, we develop a motivating example. The view is a TGV "restaurants_employees" defined by the XQuery and TGV shown in Figure 4. The query listed below retrieves all the employees’ names with restaurant identifier for employees of age greater than 25.

```
for $i in collection("restaurants_employees")//restaurant
where $i//employee/age > "25"
return
  <employee>
     {$i//employee/name}
     <restaurant id="{$i/@id}"/>
 </employee>
```

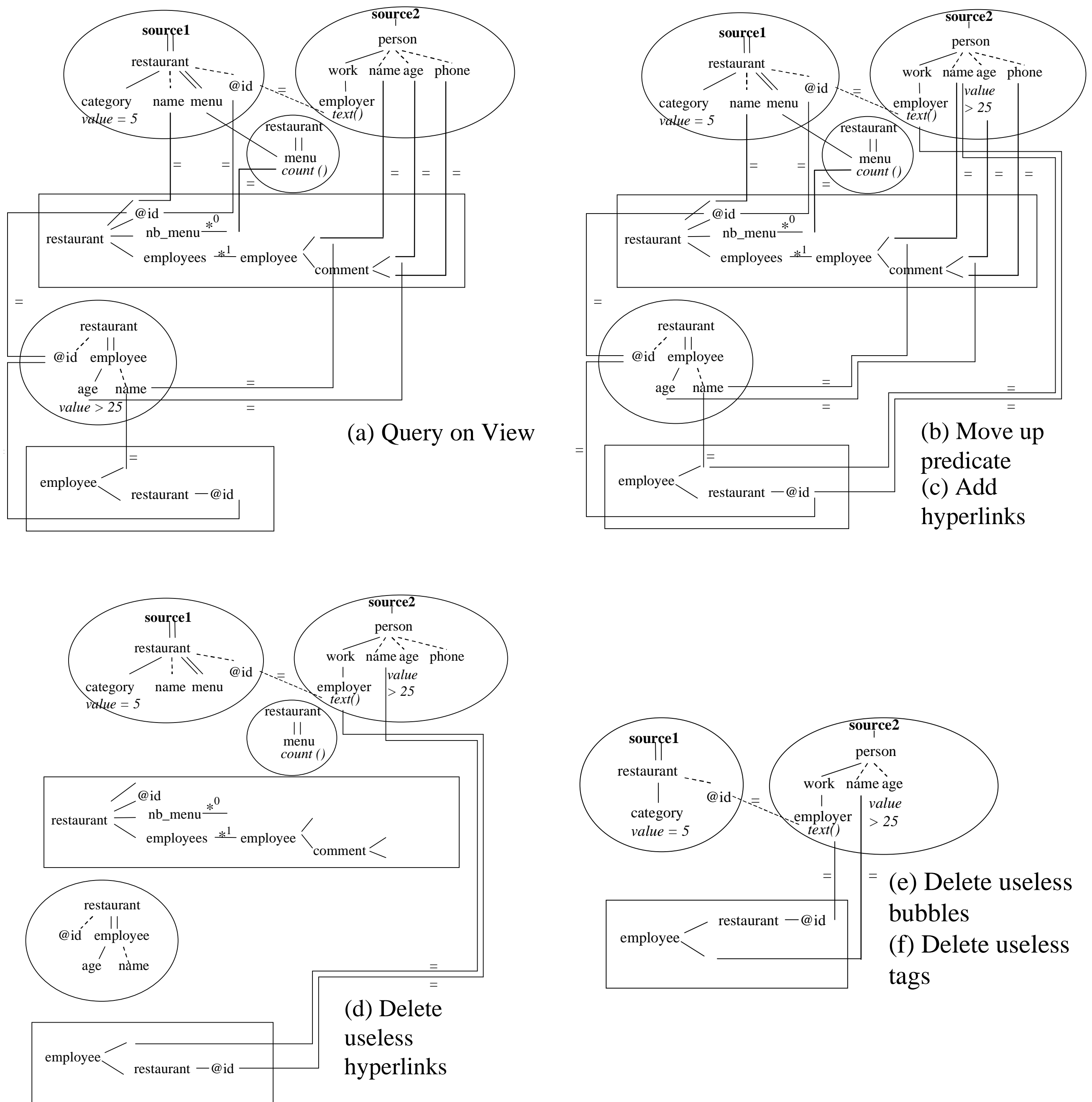

Figure 7. TGV view and logical query optimization

Figure 7 (a) represents the query applied on the view defined in Figure 4. The figure shows the first TGV derived from Figure 4, an ITP modeling the query on the view RTP, and finally a second RTP for the reconstruction of the result. Figure 7 (b) results from the application of rule 3 defined above allowing moving up a predicate along equality hyperlinks. A good heuristics consist in delegating if possible selection to the sources to decrease the

size of downloaded results. So, the predicate "age>25" is set on the STP "**source2**". Figure 7 (c) results from adding new equality hyperlinks by applying rule 7 and removing useless hyperlinks by applying rule 6. The added hyperlinks are derived by transitivity from different equal-labeled hyperlinks. From the goal (query RTP), we follow the hyperlinks to the STPs, as much as possible. First, the employee name can be linked to the "**source2**" person name directly using an equal-labeled hyperlink. Second, following the hyperlinks from the attribute @id to the "**source1**" STP on restaurant "*@id*" attribute, it is possible to infer another equality hyperlink. But the equality hyperlink modeling the join between the two STPs can be used to add an equal-labeled hyperlink from the @id attribute of the RTP to the *employer/text()* value of the STP "**source2**". The Figure 7 (d) shows a TGV with useless hyperlinks removed. In general, to simplify a TGV, redundant hyperlinks are deleted. For this purpose, our algorithm starts from the STP and only keeps useful hyperlinks that connect with an STP or the nearest ITP of an STP. All others hyperlinks become useless due to the fact that they are no more connected to the final RTP. Thence, the optimization algorithm deletes non-constraining hyperlinks recursively from the first ITP up to reaching an STP. The result is a graph with linked and unlinked bubbles as shown in Figure 7 (d).Figure 7 (e-f) represents the final optimized TGV. Bubbles without hyperlink have been removed by applying rule 2 (step e), and useless tags also by applying rule 9 (step f). Note that the bubble source1 can finally be removed if the hyperlink is not mandatory.

5. CONCLUSION

In this paper, we propose a framework for distributed XQuery processing in a mediator. The runtime engine of the mediator is based of an extended relational algebra for XML, called the XAlgebra. This framework brings out a useful tool to translate XML queries in optimized query execution plan. It can be seen as an extension and adaptation of the GTP model [5]. Our model tries to represent XRelation as bubbles including a TPQ and to link bubbles by hyperlinks modeling join predicates and extractions. The model is intuitive and is inspired from a visual query language [4]. Our specific approach proposes an algebraic representation (TGVs) and transformation rules. The optimization algorithms simply apply these transformation rules and could use cost models [10].

The important difficulties of XQuery that are optional results (null values), functions, multiplicity and nesting are somehow integrated in the model. TGV transformations are introduced to model current optimization heuristics. The model lands itself towards making simple some complex optimizations, e.g., transitivity of equality, query simplification, capabilities handling, and query through views. Further work remains to be done to better formalize the TGV transformations and direct them using a rule-based optimizer.

Another important aspect of our work is that the XLive mediator federates known source of data, whose schema can be declared at connection time. Thus, XRelation schemas are determined at query compile time. In Web Service mediation, sources may be messages with unknown and varying schemas from message to message.

Fortunately, queries are well identified and all meta-information derived from the query are known at compile time. We plan to integrate some message directed constructor to build source bubbles, so that schema-varying sources may be handled in a future version of XLive.

6. REFERENCES

[1] Abiteboul S. *On Views and XML*. PODS: 1999, pp 1-9, Symposium on Principles of Database Systems, May 31 - June 2, 1999, Philadelphia, Pennsylvania.

[2] Amer-Yahia Sihem, Cho SungRan, Lakshmanan V. S. Laks, Srivastava Divesh: *Tree pattern query minimization.* VLDB J. 11(4): 315-331 (2002).

[3] Baru C., Gupta A., Ludaescher B., Marciano R., Papakonstantinou Y., and Velikhov P.. *XML-Based Information Mediation with MIX*. In Demo Session, ACM-SIGMOD'99, Philadelphia, PA, 1999.

[4] Braga D., Campi A.: *A Graphical Environment to Query XML Data with XQuery*. WISE, pp 31-40, 2003.

[5] Chen Z., Jagadish H.V., Lakshmanan L.V.S., Paparizos S.. *From Tree Patterns to Generalized Tree Patterns: On efficient Evaluation of XQuery*. Very Large Data Bases 2003, pp 237-248, Germany Sept 2003.

[6] Christophides Vassilis, Cluet Sophie, Siméon Jérôme: *On Wrapping Query Languages and Efficient XML Integration*. SIGMOD Conference 2000: 141-152

[7] Dang-Ngoc T.-T. and Gardarin G. *Federating heterogeneous data sources with XML*. In Proc. of IASTED IKS Conf., 2003.

[8] Fernandez M.F., Morishima A., and Suciu D. *Efficient evaluation of XML middle-ware queries*. In SIGMOD '01, May 2001.

[9] Galanis L., Viglas E., DeWitt D.J., Naughton J.F. and Maier D.. *Following the Paths of XML Data: An Algebraic Framework for XML Query Evaluation*. 2001. Available at http://www.cs.wisc.edu/niagara/papers/algebra.pdf

[10] G. Graefe, W. McKenna. The Volcano Optimizer Generator: Extensibility and Efficient Search. In Proceeding of the 12th International Conference on Data Engineering, 1993, 209-218

[11] Jagadish H.V. et al. TAX: *A Tree Algebra for XML.* pp. 149-164, DBPL 2001.

[12] Manolescu I., Florescu D., Kossmann D. *Answering XML Queries over Heterogeneous Data Sources*, 27th Intl Conf VLDB, Roma, Italy, 2001, p.241-250.

[13] Naacke H., Gardarin G., Tomasic A. *Leveraging Mediator Cost Models with Heterogeneous Data Sources*. ICDE pp. 351-360, 1998.

[14] Papakonstantinou Y., Borkar V., Orgiyan M., Stathatos K., Suta Lucian, Vassalos V., Velikhov P. *XML queries and algebra in the Enosys integration platform*, Data Knowl. Eng. 44(3): 299-322 (2003).

[15] M. Roth, P. Schwarz. *Don't Scrap It, Wrap it! A Wrapper Architecture for Legacy Data Sources*. Proc. VLDB Conference, 1997.

[16] Sannella, M. J. *Constraint Satisfaction and Debugging for Interactive User Interfaces.* Ph.D. Thesis, University of Washington, Seattle, WA, 1994.