XAL: an Algebra for XML Query Optimization

Flavius Frasincar **Geert-Jan Houben** **Cristian Pau**

Department of Computer Science
Eindhoven University of Technology
P.O. Box 513, NL-5600 MB, Eindhoven, the Netherlands
Email: {flaviusf, houben, cpau}@win.tue.nl

Abstract

This paper proposes XAL, an XML ALgebra. Its novelty is based on the simplicity of its data model and its well-defined logical operators, which makes it suitable for *composability*, *optimizability*, and *semantics definition* of a query language for XML data. At the heart of the algebra resides the notion of *collection*, a concept similar to the mathematician's monad or functional programmer's comprehension. The operators are classified in three clusters: *extraction* operators retrieve the needed information from XML documents, *meta*-operators control the evaluation of expressions, and *construction* operators build new XML documents from the extracted data. The resulting algebra has optimization laws similar to the known laws for transforming relational queries. As a consequence, we propose a heuristic optimization algorithm similar to its relational algebra counterpart.

Keywords: XML, query language, query algebra, query optimization

1 Introduction

The Web is the major platform for information exchange. It is a huge database supported by theories and technologies in their infancy. In order to contribute to the formal foundation of the Web, the database theory has embarked upon a fascinating journey of rediscovery (Vianu 2001). One of the spawned Web technologies is XML, a popular standard to encode Web data. A query language that exploits the structure of XML is a hot research topic. An XML algebra provides a solid ground to define the semantics of a query language, to analyze its power of expression, and to perform query optimizations. There are a lot of proposals for XML query languages (e.g. W3C XQuery, XSLT, XMAS, XQuery, Quilt, XML-GL, XML-QL, XQL, YATL, Lore, UnQL, XDuce, YAXQL etc.) and a few for XML query algebras (e.g. W3C query algebra, XOM algebra, SAL algebra, Beech et al. algebra, OPAL algebra, YATL algebra, Lore algebra etc.).

This paper proposes XAL (XML ALgebra), an algebra that does one step further compared with the existing algebras by providing a heuristic XML query optimization algorithm. The simplicity of its data model and its well-defined logical operators facilitate the definition of optimization laws. The proposed query optimization laws are inspired by relational algebra optimization laws, monad applications to nested relational algebra optimization laws, and object-oriented query optimization.

The remainder of the paper is organized as follows. Section 2 gives an overview of some XML query algebras. Section 3 provides the data model. Section 4 contains the algebra operators. Section 5 presents the equivalence laws and a heuristic algorithm for query optimization. An example illustrating how to apply the proposed heuristic algorithm for a query expressed in XQuery is given in Section 6. Finally, Section 7 concludes the paper.

2 Related Work

The industry and research communities developed algebras for XML queries. We briefly present a few of them underlining their support for optimization.

The Lore algebra (McHugh & Widom 1999), developed at Stanford, has an idiosyncratic set of logical and physical operators. It uses cost based optimization to efficiently evaluate path expressions. The specific nature of the logical operators (built for the defined physical operators) makes it difficult to apply them to a different system.

The algebra from (Beech, Malhotra & Rys 1999) provides a logical model without considering physical operators. Unfortunately no optimization strategies are presented. Its simplicity and genericity inspired us in building XAL.

From data integration research emerged the YATL (Yet Another Tree-based Language) algebra (Christophides, Cluet & Simeon 2000), inspired by relational algebra (Ullman 1989) and object algebra (Cluet & Moerkotte 1994). Despite the optimization techniques proposed for its operators it is difficult to use it in the XML query context as it is using an idiosyncratic data model, different than the one from W3C (Fernandez & Robie 2001).

The W3C XQuery 1.0 Formal Semantics (Fankhauser, Fernandez, Malhotra, Rys, Simeon & Wadler 2001) (based on the algebra from (Fernandez, Simeon & Wadler 2001)) is a set of well-defined operators and optimization rules. It defines an unorder operator on forests (to make joins commute), but doesn't provide any details how to use it in an optimization algorithm.

The XOM (eXtensible Object Model) algebra (Zhang & Dong 1999) is a complete and closed algebra composed of six object operators. It does not provide any optimization support. The OPAL (Ordered list Processing ALgebra) algebra (Liefke 1999) proposes an XML data model based on lists. Finite state automata are used for processing and optimizing queries. The SAL (Semistructured ALgebra) (Beeri & Tzaban 1999) algebra has logical operators with a limited expressive power and support for optimizations.

In (Buneman, Fan, Simeon & Weinstein 2001) path constraints implications can be used for query optimizations. As an example, path constraints expressing inverse relationships between two elements reduce two consequent iterations to just one iteration. The algebra of (Christophides, Cluet & Moerkotte 1996) has optimization techniques based on regular (or generalized) path expressions.

Most of the algebras appear to be a mix of useful but rather ad-hoc features. In the quest for the expressibility of complex queries some of them have operators not even closed under composition (Vianu 2001). In XAL the mismatching between XML and databases is reduced by accommodating a mix of ordered and unordered operators that supports query optimization.

3 Data Model

Each XML document (Bray, Paoli, Sperberg-McQueen & Maler 2000) can be represented as a rooted connected directed graph both cyclic or acyclic with a partial order relation defined on its edges. The vertices and edges are characterized by specific properties.

Formally, the data model is

$$\begin{aligned}
G &= (V, E, O, root),\ root \in V \\
V &= V_{element} \cup V_{int} \cup V_{string} \cup V_{ID} \cup V_{IDREF} \cup \dots \\
E &= E_E \cup E_A \cup E_R \cup E_D \\
O &= \bigcup_{p \in V} ((v_1, v_2, \dots v_n) | v_i \in V,\ parent(v_i) = p)
\end{aligned}$$

The set V represents the element and value vertices. Vertices are of type *element* or *simple* (*int*, *string*, ID, $IDREF$, ...) (Fallside 2001). There is exactly one vertex that represents the top element. As each element vertex has a parent in our model we introduce an external (fictitious) vertex called *root*. The root vertex has as its unique child the top element vertex. The edge that connects the root with the top element vertex is labeled with the top element's name.

Each vertex has two basic properties as shown in Table 1. Derived properties are obtained using edge properties. Table 2 presents the derived properties of vertices. The name of a vertex is the same as the incoming element containment edge name, i.e. the name of the element that it represents.

basic property	result for *element* vertex	result for *simple* vertex
value	vertex identifier	value
type	*element*	type of value

Table 1: Basic properties for vertices

derived property	result
name	name of the vertex
parent	parent vertex (via E edge)
parentedge	incoming E edge
childelements	outgoing E edges
attributes	outgoing A edges
references	outgoing R edges
data	outgoing D edge

Table 2: Derived properties for *element* vertices

The set E corresponds to the directed edges. There are four types of edges: E, for *element containment* edges, A for *attribute edges*, R for *reference* edges, and D for text *data* edges. Element containment edges model element aggregation relationships, attribute edges represent the relationships between element and its attributes, and data edges stand for pointers to text data included in an element. Reference edges do not have an equivalent XML structure, they are virtual edges that keep the semantics of the internal reference relationship between elements (based on matching $IDREF(S)$ with ID attribute values). One could define a lexical view of the document in which id-related values are treated as textual strings, and a semantic view that defines explicitly the reference edges between matching elements (Zhang & Dong 1999). The rooted graph defined by the lexical view of the document is acyclic, i.e. a tree.

Each edge has four basic properties as shown in Table 3. Derived properties are obtained using the order relationship inherited from the implementation. Only for element containment and data edges the order is meaningful in the data model. Table 4 presents the derived properties of edges.

basic property	result
name	name of the edge
type	E, A, R, D
parent	source vertex of the edge
child	target vertex of the edge

Table 3: Basic properties for edges

derived property	result
next	following sibling edge
previous	preceding sibling edge

Table 4: Derived properties for E and D edges

The name of the edge is the element name (edge of type E), the attribute name (edge of type A), the ID attribute name (edge of type R), or the string "data" (edge of type D) that it represents.

E and D edges having the same parent are ordered in a list by the position in which they appear in the XML document. The set O is the union of these lists, one for each parent element. For A and R edges the order is not defined in the data model, even so there is always an implementation order (the order in which they appear in the document).

In the data model we neglect comments, processing instructions, and namespaces. The XML document is modeled after the entities are resolved, so we also do not consider them. XML fragments have similar structure as XML documents, thus they fit well in the same model.

4 XAL

XAL defines logical operators suitable for *composability*, *optimizability*, and *semantics definition* of an XML query language. The operators are independent of the underlying storage representation. There are three kinds of operators: *extraction* operators that retrieve the needed information from XML documents, *meta*-operators that control the evaluation of expressions, and *construction* operators that build new XML documents from the extracted data.

The chosen operators are powerful enough to formally define the semantics of a large class of queries expressed in XQuery (Chamberlin, Florescu, Robie, Simeon & Stefanescu 2001). A query optimization algorithm for XAL will have immediate benefits for the optimization of XQuery expressions. Section 6 exemplifies XQuery to XAL translation and the application of a heuristic query optimization algorithm on the resulted XAL expression.

The general form of the operators is

$$o[f](x_1, x_2, \dots x_n : expression)$$

For binary operators we also use the infix notation

$$(x : expression)\ o[f]\ (y : expression)$$

The unary operators evaluate the input to a collection of vertices and use an implicit *map* operator to compute the result. This means that the variable x is bound to each vertex in the input collection and for each such binding $f(x)$ is evaluated (f is a function operating on vertices). The semantics of the operator o defines how the partial result (result for one variable x binding) is computed from $f(x)$. The operator result is built by concatenating all the partial results. In the operators general form, f is not always present, there are operators that do not define f but use directly x (instead of $f(x)$) in computing the result. In this case the implicit *map* is absent. An n-ary operator can be transformed into a unary one by considering its input of sequence type.

An important feature of the extraction operator algebra is its *closedness*. All the extraction operators take as input collections and they return collections. As a consequence extraction operators can be composed with each other providing a rich power of expression. Compared with relational algebra the proposed one is more *flexible*: there are operations (e.g. union) working on collections containing elements of different types, a feature that is not present in relational algebra.

Collections are characterized by the *order* property. They represent a generalization of the concepts of lists, sets, and bags. The collection concept is similar to the mathematician's monad (Moggi 1989) or functional programmer's comprehension (Wadler 1987). While the monad is defined over a certain type M, we do allow for collections of arbitrary types of elements. The monad triplet of functions $(map^M, unit^M, join^M)$ can be defined for any of the considered collections. XAL has a *map* operator and a *join* (*concatenate*) operator called *union*. In XAL we do not have a unit operator as a vertex is written in the same way as a collection containing only this vertex. Monads and comprehensions are equivalent, one can produce a comprehension out of a monad and a vice-versa (Wadler 1992). Viewing collections as monads enables the application of monad laws for query optimization.

4.1 Extraction Operators

The extraction operators retrieve the needed information from the input XML documents. They return collection of vertices from the original XML graphs.

As the binary extraction operators use vertex comparison the notion of vertex equality has to be defined. Two vertices are equal if they have the same *value*. The binary extraction operators are defined in a similar way as their relational counterparts but taking in account the collection order in their operands.

For optimization purposes all binary extraction operators have set-like variants that work on unordered input collections and result in an unordered collection. In Section 5, we use this set-like variants in order to exploit their commutativity property for query optimization.

4.1.1 Projection

The projection operator is similar to the path navigation in XPath (Clark & DeRose 1999). The input is a collection of vertices and the operator follows the edges of a given *type* and *name*. It produces a collection of vertices that represent the targets of the followed edges. The order of the output collection depends on the order of the input collection.

The general form of the projection operator is

$$\pi[type, name](e : expression)$$

For *type* one can use: E, A, R, D or combinations of these separated by disjunction $|$. A regular expression over strings is used for *name*. The string that matches all names is denoted $\#$.

Example 1
$\pi[E, (P|p)ainter[s]\#](e)$ produces all the target nodes of element containment edges that have names starting with $Painter$, $painter$, $Painters$, or $painters$, and that originate from the vertices in e.

4.1.2 Selection

The selection operator is defined in a similar way as its corresponding relational operator. It takes as input a collection of vertices and checks for each vertex if the condition is valid. The vertices that fulfill the condition are gathered in the result. The output collection maintains the order of the input collection.

The general form of the selection operator is

$$\sigma[condition](e : expression)$$

In the condition (boolean function) one can use projection operators and constants as operands, comparison operators ($=, >=, <=, <, >, <>$), and logical operators (and, or, not).

Example 2
$\sigma[\pi[A, name] = \text{“}Dali\text{”}](e)$ selects all element vertices that have an attribute called *name* with the value “*Dali*”. The quotes are used to indicate that we perform a string comparison. For numbers the quotes are absent. By $\pi[A, name]$ we mean in fact $value(\pi[A, name](e))$.

4.1.3 Unorder

The unorder operator transforms a collection into an unordered one. Duplicates from the original collection are preserved. Unorder collections enables the definition of set-like (commutative) binary operators.

The general form of the unorder operator is

$$\chi(e : expression)$$

4.1.4 Distinct

The distinct operator removes duplicates from a collection. The output collection maintains the order of the input collection if present.

The general form of the distinct operator is

$$\delta(e : expression)$$

4.1.5 Sort

The sort operator sorts a collection based on a given value expression. The value expression is applied for each vertex in the input collection and the original vertices are ordered based on the computed value vertices.

The general form of the sort operator is

$$\Sigma[value\_expression(e)](e : expression)$$

If the elements to be sorted are value vertices, they can be sorted according to their values by not specifying the *value_expression*.

Example 3
$\Sigma[\pi[A, name]](e)$ orders alphabetically the input element vertices by the value of their *name* attribute.

4.1.6 Join

The join operator takes as input two collections on which it performs a cartesian product. The cartesian product is defined as two loops: the external loop traverses the left input collection and the internal loop traverses the right input collection. The pairs that fulfill the join condition form virtual vertices that have as outgoing edges, first the outgoing edges of the vertex from the left input collection, and then the outgoing edges of the vertex from the right input collection edges preserving the original edge order. In case that an input vertex is a value, a data edge is added between the virtual vertex and the value. Virtual vertices do not have the parent defined, which is a natural consequence of gluing vertices together. They are element vertices built by the system using the *vertex* operator described in Section 4.3.1. The result of the join operator is a collection of virtual vertices. The output collection is ordered if the input collections are also ordered.

The general form of the join operator and its equivalent expression are

$$(x : expression) \bowtie [condition]\ (y : expression) = \sigma[condition](x \times y)$$

All the operands and operators used in the condition of selection can be used also here. The join condition uses two variables compared with the selection condition that uses only one variable because the join operator is binary and the selection operator is unary.

Example 4
$(x : \pi[E, person](people)) \bowtie [\pi[A, id](x) = \pi[A, name](y)](y : \pi[E, painter](painters))$ pairs person and painter vertices in virtual vertices based on the equality of the *id* attribute value of a *person* and the *name* attribute value of a *painter*.

4.1.7 Cartesian Product

The cartesian product is a particular case of the join operator where the join condition is *true*.

The general form of the cartesian product operator is

$$(x : expression) \times (y : expression)$$

4.1.8 Union

The union operator combines two input collections into a single collection. The result is a collection that contains first the left input collection, then the right input collection preserving the original vertex order. The union operator can be easily generalized to an n-ary one.

The general form of the union operator is

$$(x : expression) \cup (y : expression)$$

In the definition of union, difference, and intersection operators we do not impose the "union compatible" constraint from relational algebra, feature that enables flexible XAL expressions.

4.1.9 Difference

The difference operator returns the vertices that exist in one collection but do not occur in the second collection. The result is a collection that preserves the vertex order from the first input collection.

The general form of the difference operator is

$$(x : expression) - (y : expression)$$

4.1.10 Intersection

The intersection operator returns the vertices that exist in both collections. The result is a collection that preserves the vertex order from the first input collection.

The general form of the intersection operator is

$$(x : expression) \cap (y : expression)$$

4.2 Meta-operators

The meta-operators control the evaluation of expressions. They are not real operators in the sense that they extract or construct some data elements. They are used to express repetition at input or operator level.

4.2.1 Map

The map operator applies a given function to each element of the input collection. The function results are concatenated in the output collection.

The general form of the map operator and its equivalent expression are

$$map[f](e : expression) = union(f(e_1), f(e_2), ...f(e_n))$$

where the expression e is evaluated to the collection $(e_1, e_2, ..., e_n)$.

As explained at the beginning of this section, all unary extraction operators (with f defined) have an inherent *map* operator associated to them

$$op[f](e) = map[f](e : expression)$$

In the construction phase, the *map* operator is used explicitly to iterate not just over collection of vertices but also collection of edges.

4.2.2 Kleene Star

The Kleene star operator repeats a given function possibly infinite times starting with a given input. At each iteration the results of the function are added to the next function input. Compared to the map operator the Kleene star operator includes in the result the input collection.

The general form of the Kleene star operator and its equivalent expression are

$$*[f](e : expression) = e + f(e) + ... + f(f(...(f(f(e))))) + ...$$

In the equation above the + operator is a set union. If after an iteration the output is the same as the input, a fix point is reached and there is no need to continue the repetition. A simple rule to enforce termination is that f gets descendants from an input element. Since every element has a finite number of subelements we avoid in this way infinite loops. The order of the result collection depends on the order of the input collection and the order implied by applying recursively the function.

A variant of the Kleene star operator specifies the exact number of times, n, the function should be applied. If $n = 0$ the output is the same as the input.

The general form of the Kleene star variant that specifies the number of repetitions is

$$*[f, n](e : expression)$$

Example 5
Suppose there is an XML document that gives in a hierarchical way the "influenced by" relationship between painters. We assume that there

are no loops in the graph given by the "influenced by" relation. A painter has as subelements painters that influenced him, these child painters have as subelements painters that influenced them etc. $\pi[A, name](*[\pi[E, painter]](root))$ gives the names of all painters linked by the "influenced by" relationship.

As shown in the Example 5 the Kleene Star operator is able to compute the transitive closure of a relation (or graph). Enabling recursion XAL uses the full XML power, going beyond the well known limits (not powerful enough to compute the transitive closure) of relational algebra.

4.3 Construction Operators

In querying XML data, we construct an XML document from given XML documents. With the extraction operators we can extract vertices from the given XML documents. In order to express the resulting XML document, we need operators to construct the structure of that XML document. The so called construction operators are applied on the collections of vertices retrieved in the extraction phase, and they create new vertices and edges that together express the new XML document. Note that there is no need to create reference edges as they are implicitly built by adding attribute edges for ID and $IDREF(S)$ values that reference each other.

4.3.1 Create Vertex

The create vertex operator adds a new vertex to the graph. It takes as input the *type* and *value* of the new node. The operation returns the newly created vertex.

The general form of the create vertex operator is

$$vertex[type](value)$$

For *element* vertices the *value* of the new node is the input of the system's new id generator (*nig*) skolem function which will create a unique new object id for a given input id. The function *nig* is injective, i.e. an existing id is always mapped to the same id. A *null* input *value* will result always in the creation of a new vertex.

Example 6
$vertex[element](null)$ creates a vertex of type *element* which has a new *value* (id) given by the system, and $vertex[string](\text{"}Dali\text{"})$ creates a vertex of type *string* and *value* "*Dali*".

4.3.2 Create Edge

The create edge operator adds an edge to the graph. It takes as input the vertices *parent* and *child* which the edge connects and the *type* and *name* of the new edge. The operation returns the child vertex.

The general form of the create edge operator is

$$edge[type, name, parent](child)$$

Before inserting a new edge of type E, the resulting graph is checked if it maintains the XML constraints (e.g. absence of loops defined just by element containment edges). If the XML constraints are not fulfilled, the create operation is unsuccessful.

Example 7
$edge[E, painter, vertex[element](null)](vertex[string](\text{"}Dali\text{"}))$ creates an edge of type E (element containment edge) with name *painter* between the vertices defined in Example 6.

Example 8
$map[edge[E, result, root]](e)$ where $root = vertex[element](null)$ groups in a new tree previously selected vertices (nodes) from collection e.

4.3.3 Copy Examples

Copying vertices, edges, and complete graphs are useful construction examples.

Example 9
Copying a vertex v means creating a new vertex with the type and value of the original vertex: $vertex[type(v)](value(v))$.

Example 10
Copying an edge e, involves copying also the parent and children vertices: $edge[type(e), name(e), v_p](v_c)$ where $v_p = vertex[type(parent(e))](value(parent(e))$, and $v_c = vertex[type(child(e))](value(child(e)))$.

Example 11
Copying a complete graph starting from the vertex v can be done by copying all the graph edges and their child vertices:

$$\begin{aligned}&map[edge[type(e), name(e),\\ &\qquad vertex[type(parent(e))](value(parent(e)))\\ &\qquad](vertex[type(child(e))](value(child(e))))\\ &\quad](e)\end{aligned}$$

where $e = *[parentedge(\pi[E|A|D, \#](child(x)))]$ $(x : parentedge(\pi[E|A|D, \#](v))))$.

5 Query Optimization

The main factor in the execution cost of algebra expressions is the iteration (explicit or implicit map operator) over collections. The proposed set of optimization laws aims at reducing iterations size for the data extraction expressions. The proposed laws are based on monad laws (Fankhauser et al. 2001) and relational algebraic optimizations rules (Ullman 1989) which have a natural extension to the collection context. In (Beeri & Kornatzky 1993) the known laws for transforming relational queries were successfully generalized in an object algebra context. Query processing heuristics enable the qualitative justification of the proposed laws. A cost model is outside the scope of this paper.

In the sequel we denote by $e(v)$, an expression that uses in its formula the variable v. The symbol o represents the expression composition. The empty collection is denoted by ().

Law 1 (Left unit)
If e_1 is of unit type (singleton collection), then

$$e_2(e_1) \quad = \quad e(v) \Big|_{v := e_1}$$

Law 2 (Right unit)
If e_2 is the identity function, i.e. $e_2(v) = v$, then

$$e_2(e_1) = e_1$$

Law 3 (Associativity)

$$e_1 \ o \ (e_2 \ o \ e_3) = (e_1 \ o \ e_2) \ o \ e_3$$

Law 4 (Empty collection)
If e_2 is the empty function, i.e. $e_2(v) = ()$, then

$$e_2(e_1) = ()$$

Law 5 (Decomposition of $\bowtie$)

$$e_1 \bowtie [condition]\ e_2 = \sigma[condition](e_1 \times e_2)$$

Law 6 (Decomposition of π)
If name is a regular expression that can be decomposed in several regular expressions $name_1$, ... $name_n$, and e is an unordered collection, then

$$\pi[name](e) = \pi[name_1](e) \cup ... \pi[name_n](e)$$

Law 7 (Cascading of σ)

$$\sigma[c_1 \wedge ... c_n](e) = \sigma[c_1](...(\sigma[c_n](e))...))$$

Law 8 (Commutativity of σ)

$$\sigma[c_1](\sigma[c_2](e)) = \sigma[c_2](\sigma[c_1](e))$$

Law 9 (Commutativity of σ with π)
If the condition c involves solely vertices that have incoming edges named by the regular expression name, then

$$\pi[name](\sigma[c(\pi[name])](e)) = \sigma[c](\pi[name](e))$$

Law 10 (Commutativity of σ with $\times$)
If the condition c involves solely vertices from e_1, then

$$\sigma[c](e_1 \times e_2) = \sigma[c](e_1) \times e_2$$

Law 11 (Commutativity of σ with $\cup$, $\cap$, $-$)
If θ is one of the set operators: $\cup$, $\cap$, and $-$, then

$$\sigma[c](e_1\ \theta\ e_2) = \sigma[c](e_1)\ \theta\ \sigma[c](e_2)$$

Law 12 (Commutativity of $\cup$, $\cap$, $\times$)
If θ is one of the set operators: $\cup$, $\cap$, and $\times$, and e_1 and e_2 are unordered collections, then

$$e_1\ \theta\ e_2 = e_2\ \theta\ e_1$$

Law 13 (Commutativity of π with $\times$)
If name is a regular expression that can be decomposed in two regular expressions $name_1$ and $name_2$, $name_1$ involves solely vertices in e_1, and $name_2$ involves solely vertices in e_2, then

$$\pi[name](e_1 \times e_2) = \pi[name_1](e_1) \times \pi[name_2](e_2)$$

Law 14 (Commutativity of π with $\cup$)

$$\pi[name](e_1 \cup e_2) = \pi[name](e_1) \cup \pi[name](e_2)$$

Now, we can outline the steps of a heuristic algorithm that uses the above equivalence laws to transform a query tree into an optimized tree. The optimized tree is in general more efficient to execute than the initial one.

1. *Eliminate unnecessary iterations* (use Laws 1, 2, and 4).
2. *Unorder collections* (use unorder operator). Collections for which order is not relevant are unordered.
3. *Decompose joins* (use Law 5).
4. *Decompose selections* (use Law 7). Break down selections into a cascade of selections. It enables moving select operations down in the query tree. Eliminate unnecessary iterations using Step 1.
5. *Move selections down as far as possible* (use Laws 8, 9, 10, and 11). Based on the commutativity of selection with other operators, move selection down in the query tree as far as it is permitted by the selection condition. Eliminate unnecessary iterations using Step 1.
6. *Apply the most restrictive selections first* (use Laws 3 and 12). Based on the commutativity and associativity of binary operators, rearrange the leaf vertices so that the most restrictive selections apply first. As a selectivity criterion one can use the size of the collection. The most restrictive selections are the selections that produce collections with the fewest elements. Eliminate unnecessary iterations using Step 1.
7. *Decompose projections* (use Law 6). Break down projections into a union of projections. It enables moving the project operations down in the query tree. Eliminate unnecessary iterations using Step 1.
8. *Move projections down as far as possible* (use Laws 9, 13, and 14). Based on the commutativity of projection with other operators, move projection down in the query tree as far as possible. Eliminate unnecessary iterations using Step 1.
9. *Identify combined operations* (use composition laws). Identify subtrees that group operations that can be executed by a single algorithm.

6 Query Optimization Example

We illustrate the usefulness of the heuristic optimization algorithm presented in the previous section using an example. The example is based on an XML database formed from three documents:

1. A document named `"painters.xml"` that contains many `<painter>` elements; each `<painter>` element in turn contains `<id>`, `<name>`, and `<description>` subelements.
2. A document named `"paintings.xml"` that contains many `<painting>` elements; each `<painting>` element in turn contains `<id>`, `<name>`, and `<author>` subelements.
3. A document named `"catalogue.xml"` that contains information about the price of paintings. This document contains many `<item>` elements, each of which in turn contains `<paintingid>` and `<price>` subelements.

Example 12
Consider the following query Q: "Return in alphabetical order the names of the painters that have a painting over \$ 1,000,000". The painter names will appear in the `<result>` element as many times as the number of their paintings that fulfill the above condition.

In XQuery 1.0 (Chamberlin et al. 2001) the query Q can be formulated as follows:

```
<result>
{
FOR $i IN document("painters.xml")/painter,
    $j IN document("paintings.xml")/painting
                                   [author = $i/name],
    $k IN document("catalogue.xml")/item
                                   [paintingid = $j/id]
WHERE $k/price/data() > 1000000
RETURN $i/name
SORTBY ./data()
}
</result>
```

Q is composed from two parts: an extraction part which gets the right painter names and a construction part which groups the resulted painter names in the `<result>` element. Both the input and the output of the given query are XML documents. From query optimization point of view the contruction part is not interesting. For Q the contruction part is the same as in Example 8 where e represents the collection of extracted painter names. In the previous section we proposed a heuristic query optimization algorithm that can be used to transform the extraction part of the query in a more efficient one to execute. For the rest of this section we will use the term query for the query extraction part.

The semantics of the query is captured in the initial query tree from Figure 1. In order to identify unambiguously the edge names (after a joint), we prefix them by the name of the original parent vertex.

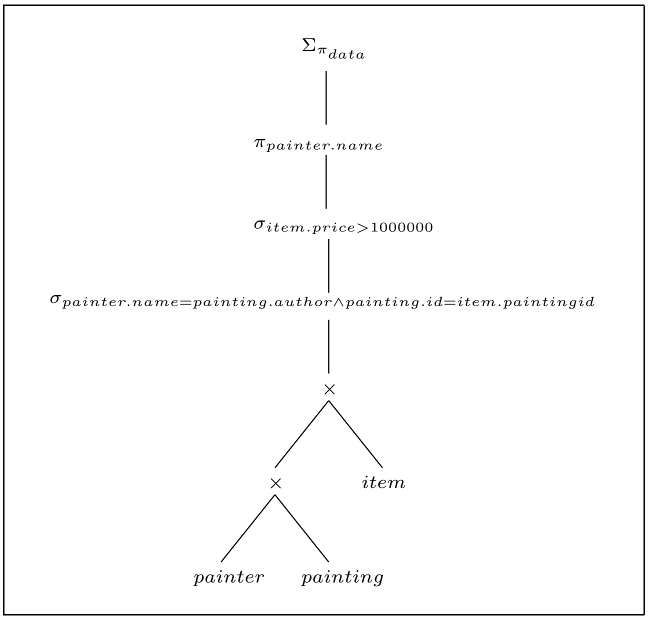

Figure 1: Initial query tree

The part of the translation scheme used to convert XQuery to XAL for our example is depicted in Table 5.

XQuery	XAL
FOR	π, σ, $\times$
WHERE	σ
SORTBY	Σ

Table 5: Translation scheme XQuery to XAL for the example

Executing the initial query tree directly gives a very large cartesian product of all *painters*, *paintings*, and *items*. But this query needs only painters that match with paintings that have a price higher than \$ 1,000,000. The original element order (given by the input files) is not significant for our query as we do sort the results alphabetically at the end. As a consequence we can use the χ (unorder) operator to transform the input collections into unordered ones and benefit from the commutativity of the XAL binary operators. First by applying Steps 2, 4, and 5 from the optimization algorithm one can get the query tree from Figure 2.

A further improvement can be realized by switching the positions of *painter* and *item* in the query tree (together with their associated selections) so that we apply the most restrictive selections first as in Step 6

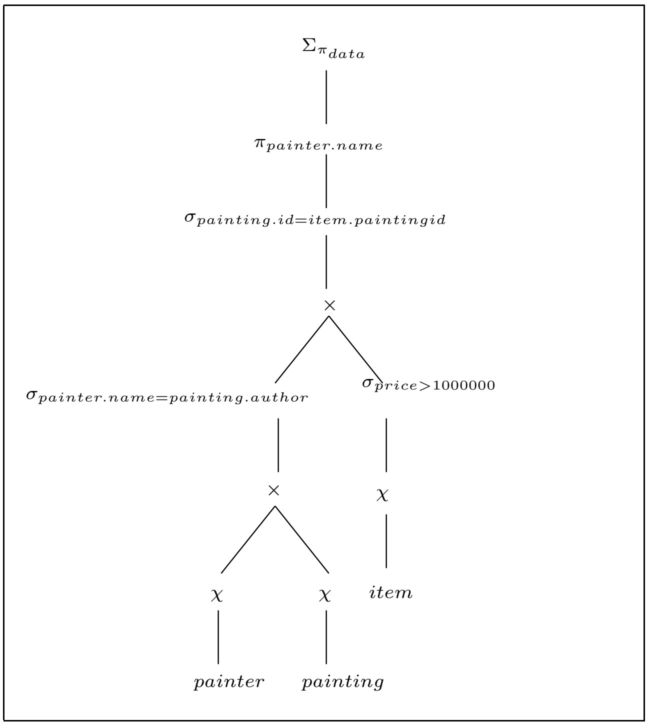

Figure 2: Query tree after first optimization

of the algorithm. The resulting query tree after this second query optimization is given in Figure 3.

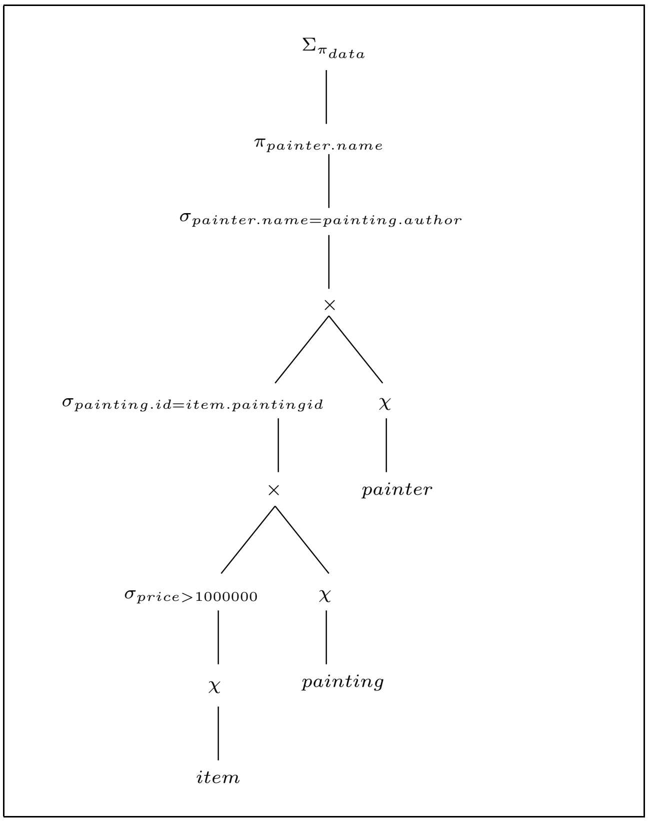

Figure 3: Query tree after second optimization

All the three query trees are equivalent. To get a better feeling why it is more efficient to execute the last query tree instead of the first and second ones we will give a quantitative dimension to our example. Suppose there are three painters: the first one has 100 paintings, the second one 150 paintings, and the last one 100 paintings. Only the first one has 20 paintings which are more expensive than $ 1,000,000. If we use the first query tree we will need to compute for the cartesian product $3 \times 350 \times 350 = 367{,}500$ elements, for the second query tree 3×350 (painters are matched to their paintings) $+ 350 \times 20 = 8{,}050$ elements, and for the last query tree 20×350 (expensive paintings are matched to their description) $+ 20 \times 3 = 7{,}060$ elements. One can notice that the last query tree is the most effective one to execute as it needs less computations compared to the first two queries.

7 Conclusion

Compared with existing XML algebras, XAL provides operators similar to the relational algebra ones which are familiar to the relational database world. It also lifts the known heuristic optimization algorithm for relational queries to the XML context providing an elegant way of doing so. XAL and its optimization laws can be used by a query optimizer for choosing efficient transformation alternatives. It can also be used for defining the semantics of an XML query language and as a comparison framework in terms of power of expression and flexibility of such query languages. As future work we will further explore XAL properties and applications. We plan also to investigate new optimizations laws that take advantage of the XML specific features (e.g. tree structure, internal references).

References

Beech, D., Malhotra, A. & Rys, M. (1999), A Formal Data Model and Algebra for XML. Available from `http://www-db.stanford.edu/dbseminar/Archive/FallY99/malhotra-slides/malhotra.pdf`.

Beeri, C. & Kornatzky, Y. (1993), ‘Algebraic Optimization of Object-Oriented Query Languages’, *Theoretical Computer Science* **116**(1&2), 59–94.

Beeri, C. & Tzaban, Y. (1999), SAL: An algebra for Semistructured Data and XML, *in* ‘Proc. ACM SIGMOD Workshop on The Web and Databases (WebDB’99)’, pp. 37–42.

Bray, T., Paoli, J., Sperberg-McQueen, C. M. & Maler, E. (2000), Extensible Markup Language (XML) 1.0 (Second Edition), Technical report, W3C. Available online at `http://www.w3.org/TR/2000/REC-xml-20001006`.

Buneman, P., Fan, W., Simeon, J. & Weinstein, S. (2001), ‘Constraints for Semistructured Data and XML’, *ACM SIGMOD Record* **30**(1), 47–45.

Chamberlin, D., Florescu, D., Robie, J., Simeon, J. & Stefanescu, M. (2001), XQuery 1.0: An XML Query Language, Technical report, W3C. Available online at `http://www.w3.org/TR/xquery`.

Christophides, V., Cluet, S. & Moerkotte, G. (1996), Evaluating Queries with Generalized Path Expressions, *in* ‘Proc. 1996 ACM SIGMOD International Conference on Management of Data (SIGMOD’96)’, ACM Press, pp. 413–422.

Christophides, V., Cluet, S. & Simeon, J. (2000), On Wrapping Query Languages and Efficient XML Integration, *in* ‘Proc. 2000 ACM SIGMOD International Conference on Management of Data (SIGMOD’00)’, ACM Press, pp. 141–152.

Clark, J. & DeRose, S. (1999), XML Path Language (XPath) Version 1.0, Technical report, W3C. Available online at `http://www.w3.org/TR/xpath`.

Cluet, S. & Moerkotte, G. (1994), Nested Queries in Object Bases, *in* ‘Proc. Fourth International Workshop on Database Programming Languages (DBPL’93)’, Workshops in Computing, Springer Science Publishers, pp. 226–242.

Fallside, D. C. (2001), XML Schema Part 0: Primer, Technical report, W3C. Available online at `http://www.w3.org/TR/xmlschema-0`.

Fankhauser, P., Fernandez, M., Malhotra, A., Rys, M., Simeon, J. & Wadler, P. (2001), XQuery 1.0 Formal Semantics, Technical report, W3C. Available online at `http://www.w3.org/TR/query-semantics`.

Fernandez, M. & Robie, J. (2001), XQuery 1.0 and XPath 2.0 Data Model, Technical report, W3C. Available online at `http://www.w3.org/TR/query-datamodel`.

Fernandez, M., Simeon, J. & Wadler, P. (2001), A Semi-monad for Semi-structured Data, *in* ‘Proc. 8th International Conference on Database Theory (ICDT’01)’, Vol. 1973 of *Lecture Notes in Computer Science*, Springer, pp. 263–300.

Liefke, H. (1999), Horizontal Query Optimization on Ordered Semistructured Data, *in* ‘Proc. ACM SIGMOD Workshop on The Web and Databases (WebDB’99)’, pp. 61–66.

McHugh, J. & Widom, J. (1999), Query Optimization for XML, *in* ‘Proc. 25th International Conference on Very Large Data Bases (VLDB’99)’, Morgan Kaufmann, pp. 315–326.

Moggi, E. (1989), Computational Lambda-Calculus and Monads, *in* ‘Proc. Fourth Annual Symposium on Logic in Computer Science (LICS’89)’, IEEE Computer Society Press, pp. 14–23.

Ullman, J. D. (1989), *Principles of Database and Knowlwdge-Base Systems*, Vol. 1&2, Computer Science Press.

Vianu, V. (2001), A Web Odyssey: from Codd to XML, *in* ‘Proc. Twenteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS’01)’, ACM Press.

Wadler, P. (1987), *The Implementation of Functional Programming Languages*, Prentice Hall, chapter 7. The author of the book is Simon L. Peyton Jones.

Wadler, P. (1992), ‘Comprehending monads’, *Mathematical Structures in Computer Science* **2**(4), 461–493.

Zhang, D. & Dong, Y. (1999), A Data Model and Algebra for the Web, *in* ‘Proc. 10th International Workshop on Database & Expert Systems Applications (DEXA’99)’, IEEE Computer Society, pp. 711–714.