A Formal Data Model and Algebra for XML

Editors:

David Beech (Oracle) dbeech@us.oracle.com
Ashok Malhotra (IBM) petsa@us.ibm.com
Michael Rys (Microsoft) mrys@microsoft.com

Requirements for XML Query

As XML becomes more popular and, in particular, becomes more popular for encoding data, a XML query language will become more important in order to facilitate the query and integration of XML encoded data without necessitating the transformation of the data into another format such as relational data.

To move towards a formalism for a XML query language, this paper presents a formal data model for XML. It shows how the components of a XML document and their interrelationships can be represented as a directed graph. Subsequently, it discusses operations on the graph that form the basis for querying and manipulating XML.

We see the following requirements for a XML query language:

- Retrieve XML documents or fragments of documents from a collection of documents based on specified selection criteria.
- The documents may have been originally authored as XML documents (*real documents*) or they may be an XML view of existing data (*virtual documents*).
- Real XML documents may be stored in the underlying repository in a fragmented fashion based on some *mapping*.
- The results from a XML query may be XML documents or collections of fragments.
- XML documents or fragments may be selected based on their structural as well as on their data content.

The following data model is a logical model and is silent on how it's components should be stored. Logical operations on the model will need to be translated to operations on the underlying storage representation before they can be executed.

Introduction

An XML document consists of elements that contain data or other elements. Each element is typed and, depending on its type, may contain one or more attributes. Child elements or sub-elements of a parent element are ordered whereas its attributes are not ordered. Attributes contain only data, i.e., they cannot contain elements nor have attributes.

Special attributes are designated as IDs. The value of each ID attribute must be unique in the document. Other special attributes are designated as IDREFs. The value of each IDREF attribute must equal the value of an ID attribute. In this way XML elements within a document can refer to each other. Attributes of type IDREFS can refer to a set of elements. Another mechanism for elements to refer to one another is to store a URI or a XLink as the data of an element. This allows elements to refer to elements outside as well as inside the document. These facilities extend XML from a pure hierarchy into a graph.

XML supports *entities* which allow special symbols to be replaced by simple text or text containing markup. In most cases, the mapping from XML into the data model occurs after entities have been resolved, so there are no entities in the data model. For large external entities that are not resolved, the reference to the entity is treated as a data value.

XML also has other features such as comments and processing instructions. These are treated as special kinds of data in the data model.

The XML Infoset Specification describes the required and optional information available from an XML document. It is, therefore, a data model for XML. While our data model mostly adheres to the XML Infoset specification, there are some difference. From our viewpoint, the Infoset has too much lexical information and is missing information about references. For example, it has information about individual characters, about white space, and about external entities that we do not represent or represent differently. For query purposes we need less lexical information i.e. we need a more abstract model. The model described in this paper also has features not present in the Infoset such as explicit information about references.

Data Model Overview

An XML document is represented by a directed graph. The graph contains two types of vertices (or nodes): vertices that represent elements of the document and vertices that represent data values.

The graph also contains three types of directed edges. A set of directed *element containment* edges, *E*, relates parent elements to their children. The children may be elements or they may be data values. A set of directed *attribute* edges, *A*, relates elements to attribute data values. A set of *reference* edges *R* relate elements to other elements they reference via IDREFs, IDREFSs, XLinks, URIs, or other reference mechanisms. These edges are present in the model in addition to the edges that relate the element to the referent data. For attributes that refer to other elements via IDREFs or IDREFSs, the additional reference edges are present only if there is enough information available to identify the attributes' referential nature (e.g., via a DTD or schema).

Finally, the ordering of child elements within a parent element is captured in a set of ordering relations *O*. The order of children connected by element containment edges is the order in which the children appear within the parent element. The order of attribute edges is not defined in the model and is implementation dependent. Multi-valued references that are represented by multiple reference edges will be ordered among themselves according to the reference ordering rules (e.g., for IDREFS in the order they were written). Even if the order is not defined in the model, there is always a default implementation order among *A* and *R* edges emanting from a single parent that will be preserved by the ordering relations.

Each XML element is represented by an element vertex with a unique, immutable, system generated identifier. The model places no restriction on the form of the identifier. In this document we refer to the identifier of the element vertex that represents the element <x/> as v_x. Multiple elements of the same type, such as children of type x, would be referred to as v_{x1}, v_{x2} etc..

Every element containment edge in *E* connects a parent vertex and a child vertex. If the parent and child are both element vertices the name of the edge is the generic identifier of the child element. Edges from parent elements to value vertices have the special name *~data*. Edges from parent elements to comment and processing instruction vertices have the names *~comment* and *~PI*, respectively.

Each element vertex must have a parent. In case of XML fragments where the outermost element has no super element a fictitious root vertex is provided for this purpose.

Attribute edges in *A* relate element vertices to attribute values, i.e., value vertices. In the special case that the attribute value refers to another element via an IDREF, IDREFS, XLink or URI, the value vertex stores the data value of the reference and one or more *R* edges connect the element to its referent. For IDREFs this requires the presence of a DTD or a XML Schema that identifies the attribute as an IDREF. Similarly, if an attribute can be identified as an IDREFS attribute an additional set of *R* edges point to a set of elements.

Edges that refer to elements outside the data model scope via a URI or XLink are given a special stub value that knows how to find the external element being referred to.

An ordering relationship is defined over the edges from a parent vertex to its children of each type.

Vertex and Edge Properties

Element vertices have as basic properties:

- *value* returns the identier for the vertex.
- *type* returns the type of the vertex (for element vertices it is currently always *element*).

In addition, the following properties can be derived based on edge and order information:

- *gi* returns the (namespace qualified) name of the vertex.
- *parent* returns the parent vertex.
- *referredby* returns the set of vertices that reference the vertex via a reference edge.
- *childelements* returns the set of all element containment edges originating at the vertex.
- *attributes* returns the set of all attribute edges originating at the vertex.
- *references* returns the set of all reference edges originating at the vertex.

Value vertices have two basic properties:

- *type* indicates the datatype of the value.
- *value*

For example, the datatype of an IDREF attribute value will be IDREF, a CDATA content of an element will be of type CDATA (or string).

Element containment edges (including *~data*, *~comment* and *~PI* edges) have four basic properties:

- *parent* the referring vertex
- *child* the vertex referred to
- *name* the name of the edge
- *type* (E in this case)

In addition, the properties

- *next* the following edge
- *previous* the preceding edge

can be derived from the ordering relation and provide the next or previous element containment edge with the same parent. Attribute edges also have the four basic properties: *parent*, *value*, *name* and *type* and the two derived properties *next* and *previous*.

Both *name* properties encode the namespace information by representing the name as a tuple consisting of the namespace definition edge and the generic identifier. In the case of qualified elements and attributes, the namespace definition edge is the edge of the attribute that defines the namespace; in the case of unqualified attribute names, it is its parent's element containment edge. In order to comply with the W3C Namespace specification section A3 that describes a model of namespace-qualified names in which a full name is a pair consisting of the namepace URI and the local part of the name, we will introduce derived properties that return the edge's full name with namespace prefix (*qualifiedname*) and its URI (*ns_uri*). Element names that are unqualified (i.e., they don't have a namespace prefix) have an empty URI, and their qualified name is equal to their simple name. Note that for reason of simplicity, we will often only write the local part of the name to identify an edge name.

Finally, reference edges have the three basic properties: *parent*, *child* and *type*. They also have another basic property *refedges*, a set of the attribute or element edges that provide the reference information (if available). The *type* property indicates that this edge is a reference and also the kind of reference it is e.g. an XLink or a URI, etc.. The *name* property of a reference edge is derived from the *refedges*' name(s). Reference edges also have the two derived properties *next* and *previous*.

The properties of the ordering relation among edges are:

- *e* which returns the current edge.
- *successor* which defines the successor of the current edge.

The order is a total order among all edges of a type, E, A or R that possess the same parent. Note that the order does not explicitly define an order among edges of different types or edges with different parent vertices. The properties:

- *predecessor*
- *first*
- *last*
- *[n]* the nth edge

can be derived from the two basic properties.

The properties of vertices and edges are summarized in the section Property Overview below.

Examples of data models produced from XML are shown in Example 1, Example 2. Example 3 Example 4 and Example 5

Algebra Overview

The query algebra provides a number of operations on the data model. These operations can be composed and used to select whole documents or parts of documents and to create models from the selected parts.

Navigation Overview

The basic navigation operation *follow* or, ϕ, starts with a set of vertices and follows edges of a given type and with a given name. There is a great deal of flexibility on how the type and name is specified. The type may be any combination of E, A or R or "all" and the name may be specified as a regular expression.

The follow operation returns a set of edges. To get the vertices reached from these edges the follow operation must be composed with the *child* operation. Similarly, follow operations may be composed with one another to follow a set of edges.

For example, the operation

$$\phi[\text{A, name}](\textit{Customers})$$

follows the attribute edge *name* of all the Customer vertices.

The *inverse follow* operation ϕ_{inv} returns all the edges of the specified type and name that lead to the specified set of vertices.

The Kleene star operation, $*$, can be applied to a follow or inverse follow operation to repeat the operation until a fixed point is reached. That is, the set of vertices does not change by repeating the operation any more.

Details of the navigation operation are discussed in the section Navigation.

Selection Overview

The selection operator, σ, allows us to express a selection over a collection. It applies a given condition to each member of the collection and returns a result collection consisting of those members for which the condition evaluates *true*.

For example, the selection

$$\sigma[\mathit{child}(\phi[\mathrm{A}, \mathrm{name}](c)) = \text{'John Doe'}](c{:}\mathit{Customers})$$

selects all the customer vertices that have a *name* attribute with value *John Doe*.

Since the selection operator takes a collection as input and returns a collection, selection operators can be composed.

Vertex and edge properties such as *value* and *type* can be used to construct selection conditions. With them the above query is more correctly expressed as:

$$\sigma[\mathit{value}(\mathit{child}(\phi[\mathrm{A}, \mathrm{name}](c))) = \text{'John Doe'}](c{:}\mathit{Customers})$$

Other standard comparison operators (such as <>, <, >, <=, >=), can be used instead of = in the selection condition. Boolean operators (*and, or, not*) can be used to create more complex conditions and navigation can be used along with selection to specify conditions on a hierarchy. Existential and universal quantification are also supported.

Details of the selection operation are discussed in the section Selection.

Joins Overview

A query runs over a collection of documents. In some cases we want to query more than one collection of documents where individual documents from different collections have some relationship to one another. These relationships are called *joins* and are expressed by *join conditions*.

A join condition takes two collection of vertices as arguments along with a condition. It evaluates the condition on a cartesian product of vertices from the two collections. For each pair for which the join condition evalues *true* a virtual reference edge is created between the two vertices. This is a virtual reference edge in that it exists only for the span of the query. During the query this edge can be used in the query specification as any other reference edge.

Details of the join operation are discussed in the section Joins.

Construction Overview

One form that a query may take is to select all documents from a collection that satify certain conditions. For this, the user can construct selection conditions from the navigation and selection operations. The power of the algebra provides the ability to create conditions of arbitrary complexity.

Other kinds of queries may want to expose components or fragments of the selected document(s). The expose and return operations are designed to support this. The *expose* operation returns the document fragments identified by navigation operations in conjunction with selection operations. The *return* operation returns *copies* of fragments identified by navigation operations in conjunction with selection operations. The algebra does not discuss the form in which the results are presented to the user.

In perhaps what will be the most common form of XML query, new XML documents will be created from fragments of selected documents. The *create vertex* and *create edge* operations are designed to support the creation of new XML documents. XML documents can be constructed by attaching edges and vertices to the root and recursively attaching edges and vertices to the attached vertices.

Other Useful Operations

The algebra provides a number of other operations to assist in the specification of queries. Among them are:

- *sort* allows the reordering of a set of edges.
- *map* applies a specified function to a collection of edges or vertices.
- *unorder* indicates that the order of a set of edges is unimportant. This facilitates optimization.
- *distinct* eliminates duplicates from a set of edges or vertices.

Putting It All Together

Users often want to create vertices and edges based upon information in a group of similar vertices or edges. The Grouping operation allows collections to be created from similar vertices and edges. This section also discusses how some of the other algebra operations discussed earlier can be used to create vertices and edges from information in these collections.

A Formal Data Model

Let graph $G = (V, E, A, R, O)$ represent the data model for XML documents. V is the set of element and value vertices in the graph (formally, $V = V_{element} \cup V_{int} \cup V_{string} \cup \ldots$). E represents the set of directed edges that express element containment in a XML document. A is the set of directed edges that represent the relationships between elements and values expressed by XML attributes. R represents the relationships between elements referenced from other elements via IDREF and IDREFS attributes in the presence of schema or DTD information, URI, XLink, or foreign keys. Finally, O represents the total order between edges of a particular class, E, A or R, that connect a parent element to its children.

Vertices V

Each XML element is represented as an element vertex $v \in V_{element}$. v has a unique, logical and immutable abstract identifier. It has to be unique to serve its role as identifying key, it is logical in the sense that the identifier is independent from the physical store (i.e., it is not a ROWID), and it is immutable to preserve relationships expressed using the identifier. The abstract identifier cannot be directly accessed by the query. The type of element vertices is *element*.

Each data value in the XML document is represented by a value vertex $v \in V_{type(v)}$. A value vertex does not have a unique identifier and has *value* and *type* properties. In the following, we will often only write the value when the type is clear from the context. E.g., ("42", *string*) will be represented as "42".

We can specify the function *vertex*(*x*) that transforms an XML element or concrete value into a vertex. If *x* is an XML element, it returns the element's abstract identifier and *element* as its type; else it returns its value and concrete datatype:

x is XML element:	$vertex(x) \in V \equiv (get\_oid(x), element) \in V_{element}$
otherwise:	$vertex(x) \in V \equiv (x, type(x)) \in V_{type(x)}$

In order to access a vertex, the vertex has to be referenced via an edge.

Derived Vertex Properties

There are several properties of a vertex that can be derived from edge and order properties.

The property $gi{:}V \rightarrow T_{name}$ returns the name of the vertex: $gi(v) \equiv name(e) \mid \exists e \in E{:}child(e)=v$. This name corresponds to the generic identifier of the XML element represented by the vertex. T_{name} denotes the name type that preserves the namespace information.

The property $parent{:}V \rightarrow V_{element}$ returns the parent element vertex that contains the vertex: $parent(v) \equiv parent(e) \mid \exists e \in E{:}child(e)=v$.

The property $referredby{:}V \rightarrow \{V_{element}\}$ returns the set of element vertices that reference the vertex via a reference edge: $referredby(v) \equiv \{parent(e) \mid \exists e \in R{:}child(e)=v\}$. Note that this property could be further refined according to the type of edge that defined reference, e.g., $referredby\_attr(v) \equiv \{parent(e) \mid \exists e \in R{:}child(e)=v \wedge refedges(e) \subseteq A\}$.

The property $childelements{:}V \rightarrow \{E\}$ returns the set of element containment edges originating at the vertex: $childelements(v) \equiv \{e \mid \exists e \in E{:}parent(e)=v\}$.

The property $attributes{:}V \rightarrow \{A\}$ returns the set of attribute edges originating at the vertex: $attributes(v) \equiv \{e \mid \exists e \in A{:}parent(e)=v\}$.

The property $references{:}V \rightarrow \{R\}$ returns the set of reference edges originating at the vertex: $references(v) \equiv \{e \mid \exists e \in R{:}parent(e)=v\}$.

In addition, we can define derived properties to navigate among the children. An example is $first\_contained\_child{:}V \rightarrow V$ that returns the vertex of the first contained subelement. It is defined as $first\_contained\_child(v) \equiv child(e) \mid \exists e \in E{:}parent(e)=v \wedge \forall o \in O{:}child(e) \neq succ(o)$. Note that many of these child related properties can also be expressed using the navigational operator introduced in the algebra.

Edges E, A, R and their Order O

Every edge $e \in E$ is a directed relationship ($name \in T_{name}$, $parent \in V_{element}$, $child \in V$) from an element *parent* vertex to a *child* vertex with the name *name*. In the case of *~data*, *~comment* and *~PI* edges, *child* is a value vertex, otherwise it is an element vertex.

Every edge $e \in A$ is a directed relationship ($name \in T_{name}$, $parent \in V_{element}$, $child \in V_{value}$) from an element *parent* vertex to a *child* vertex with the name *name*.

Every edge $e \in R$ is a directed relationship ($parent \in V_{element}$, $refegdes \in \mathcal{P}(E \cup A)$, $child \in V$) from an element *parent* vertex to an element *child* vertex that is indicated by a IDREF or IDREFS attribute in the presence of a DTD or other schema information, a XLink, foreign key reference, or URI value. *refedges* denotes the set of edges that form the basis of the reference. An IDREFS attribute or a multivalued XLink is mapped into multiple R edges, one for every element referred to.

O defines an order between edges if and only if they share the same parent and they are all of the same class i.e. all E, A or R. Formally,

$$O \equiv \{(e \in E \cup A \cup R, succ \in E \cup A \cup R) \mid parent(e) = parent(succ) \wedge (e \in E \wedge succ \in E \vee e \in A \wedge succ \in A \vee e \in R \wedge succ \in R)\}$$

where *succ* denotes the successor of *e* in the order. In case of reference edges, the order among individual references of multi-valued references is defined according to the rules of the reference mechanism. In addition, the order among the *refedges* determine the order among the different references. This implies, that there is only a total order among references that have the same type of *refedges* and the order among all reference edges is partial.

The predecessor edge in the order can be determined by $pred(x) \in E \cup A \cup R \equiv e(o) \mid \exists o \in O{:}succ(o) = x$.

Element Containment Edges E

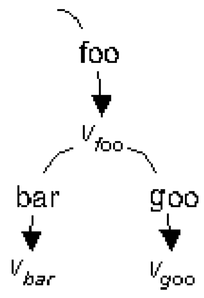
Each vertex *v* that represents an XML element *x* has exactly one incoming edge $e \in E$ such that $e = (name(x), vertex(parent(x)), vertex(x))$ where *name*(*x*) returns the name of element *x*, *parent*(*x*) returns the element that contains *x*, and *vertex*(*y*) maps the element *y* to a vertex $v \in V$ as previously defined. Thus, *vertex*(*x*) = *v*. Informally, each element vertex has an element containment edge from its parent to it.

In case of XML fragments where the outermost element has no explicit parent, an artificial root vertex is created and *parent*(*e*) is the object representing this root vertex v_{root}. The order among such outermost elements is preserved in *O*.

Example 1: The XML fragment `<foo><bar/><goo/></foo>` is represented as

E				***O***	
Edge	***name***	***parent***	***child***	***e***	***succ***
e_1	"foo"	v_{root}	v_{foo}	e_1	*null*
e_2	"bar"	v_{foo}	v_{bar}	e_2	e_3
e_3	"goo"	v_{foo}	v_{goo}	e_3	*null*

Note that *null* represents the non-existent object in all examples. Graphically, we represent the above XML graph as follows (order is implied from left to right unless otherwise indicated):



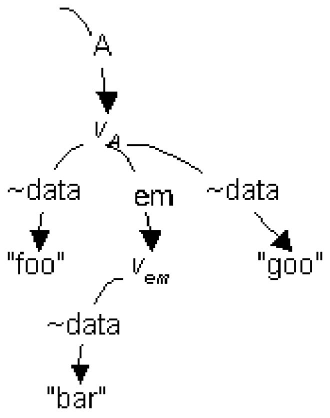
Value and Containment Edges

Since contained PCDATA might not be tagged (see Example 2) or might be contained by an element that also possesses attributes (see Example 3) the incoming edge to a value vertex is in general not an element containment edge with the name of the containing element but a special *~data* edge.

Example 2: The XML fragment `<A>foo<em>bar</em>goo</A>` is represented as

E				***O***	
Edge	***name***	***parent***	***child***	***e***	***succ***
e_1	"A"	v_{root}	v_A	e_1	*null*
e_2	*~data*	v_A	`"foo"`	e_2	e_3
e_3	"em"	v_A	v_{em}	e_3	e_5
e_4	*~data*	v_{em}	`"bar"`	e_4	*null*
e_5	*~data*	v_A	`"goo"`	e_5	*null*

Or graphically:



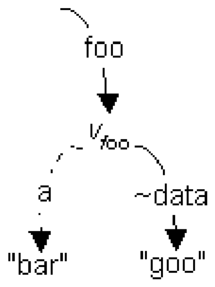
Note that element *A* has mixed content. The outgoing edges from *A* are ordered according to the occurrence of the text (data) or markup.

Example 3: The XML fragment `<foo a="bar">goo</foo>` is represented as

E				***O***	
Edge	***name***	***parent***	***child***	***e***	***succ***
e_1	"foo"	v_{root}	v_{foo}	e_1	*null*
e_3	*~data*	v_{foo}	`goo`	e_2	*null*
				e_3	*null*

A			
Edge	***name***	***parent***	***child***
e_2	"a"	v_{foo}	`bar`

Or graphically:



The ~*comment* and ~*PI* edges are treated in a manner similar to ~*data* edges. The edges that connect them to their parent appear in the order that the information appears in the parent element.

Derived Properties

The derived property $next{:}E \to E$ provides the next element edge within the same parent. It is defined as: $next(e) \equiv succ(o) \mid \exists o \in O{:}e=e(o)$. *previous* is defined in an analogous manner.

Containment edges *e*(as all other type of edges) have a derived property $type(e) = E$ that returns their edge type.

Attribute Edges A

Each XML attribute $a=(name, value, type)$ of an element <x/>, where *name* is the attribute name, *value* the attribute's value and *type* its datatype, is represented as an edge $e \in A$,such that $e= (name(a), vertex(x), value(a))$.

Reference Edges R

If the datatype of the attribute can be identified as being a reference (e.g., if it is of type IDREF, IDREFS, a foreign key, a XLink, a URI or a generated by a join operation), additional edges $\{r_i\} \in R$ are defined. In the case of references introduced through typing (such as IDREF), $r_i = (parent(e), \{e\}, refattr(child_i(e)))$ where *e* denotes the attribute edge that has the reference typed vertex as child, $child_i(e)$ denotes the *i*-th reference (an IDREFS typed attribute can have more than 1 reference). The function *refattr*(*v*) returns the vertex that is referenced by the value vertex *v*, if *v* is of a reference type (such as IDREF, HREF etc.). Formally, where ID stands for all id types and REF for all reference types.

$$refattr(v) \in V_{element} \equiv parent(a) \mid type(v) = REF \wedge \exists a \in A{:}child(a) = value(v) \wedge type(child(a)) = ID$$

Note that the definition of *R* edges is slightly different for references via element content or multi-attribute reference constructs such as multi-attribute foreign keys.

While the above works for simple XLinks (xml:link="simple"), the data model can also provides a special understanding of extended XLinks (xml:link="extended") because they actually imply a named reference originating at the extended link element instead of the element *locator*. The names of these reference edges are given by the role attributes of the locator elements (the role attribute can also become the name for the simple link). Currently the data model does not handle extended link groups.

The *type* property of reference edges indicates that the edge is a reference edge and also the kind of reference it is. It can have the following values:

- R_{id} for an ID reference
- R_{key} for a key reference
- R_{link} for a link reference
- R_{uri} for an URI reference
- R_{join} for a join reference

The use of reference edges in conjunction with joins is explained in the join section of the algebra description. Essentially, joins introduce temporary reference edges that exist only for the duration of the query.

Note that by adding the reference edges (instead of replacing the attribute or element edges), we preserve the actual values used for expressing the references. This allows us to generate the same subgraph of element containment and attribute edges regardless of whether schema information is available or not to identify ID/IDREFs. In addition, if an attribute references an element with two attributes that serve as two possible identifiers for the element such as an ID and a URI, we can preserve the exact nature of the reference, i.e., we can reconstruct the right attribute reference if we reconstruct the original XML document. Example 4 illustrates this.

Example 4: In the following, we use attr::type to indicate that the attribute is of the specified type. The XML fragment

```
<Person SSNo::ID="666-66-6666" Empno::URI="../foo.html"/>
<INS Taxpayer::IDREF="666-66-6666" />
<Company Employee::HREF="../foo.html" />
```

is represented as

E

Edge	***name***	***parent***	***child***
e_1	"Person"	v_{root}	v_{Person}
e_4	"INS"	v_{root}	v_{INS}
e_6	"Company"	v_{root}	$v_{Company}$

A

Edge	***name***	***parent***	***child***
e_2	"SSNo"	v_{Person}	`("666-66-6666", ID)`
e_3	"Empno"	v_{Person}	`("../foo.html", URI)`
e_5	"TaxPayer"	v_{INS}	`("666-66-6666", IDREF)`
e_7	"Employee"	$v_{Company}$	`("../foo.html", HREF)`

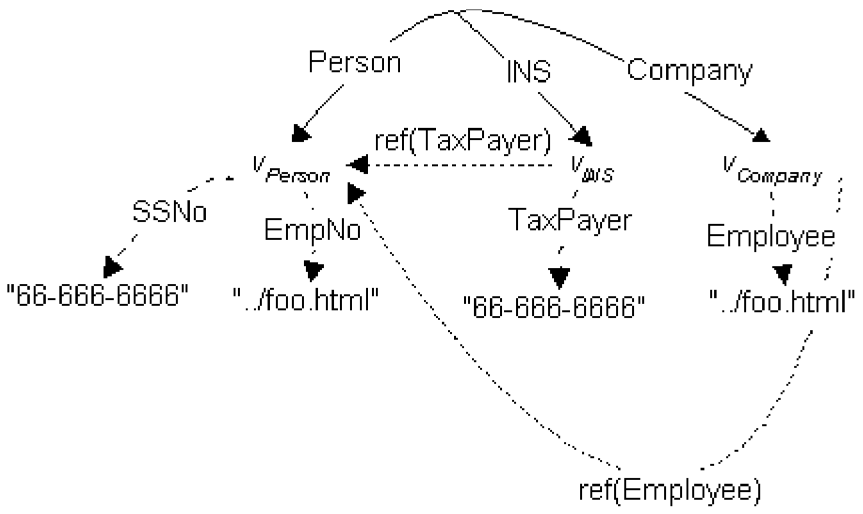
R

Edge	***parent***	***refedges***	***child***
e_8	v_{INS}	$\{e_5\}$	v_{Person}
e_9	$v_{Company}$	$\{e_7\}$	v_{Person}

O

e	***succ***
e_1	e_4
e_2	e_3
e_3	*null*
e_4	e_6
e_5	*null*
e_6	*null*
e_7	*null*
e_8	*null*
e_9	*null*

Or graphically:



This example shows two reference edges, ref(Employee) and ref(Taxpayer) (ref() is added to the graph to distinguish them from the element edges of the same name) that refer to the same v_{Person} vertex. Reference edges might point to an external XML element. If the element referred to is not in the same graph, a local surrogate object (or *stub object*) is added to the graph to provide information about the location of the external object.

While XML currently does not require that the order of attributes or the order of elements in multi-valued attributes is preserved, *O* can preserve such orderings to preserve the exact nature of the XML document (see Example 4).

Derived Properties

A reference edge possesses the derived property $name{:}R \to T_{name}$ that returns the name of the edge based on the naming scheme implied by the reference type. For example, references implied by an IDREF attribute ($type = R_{id}$) have the same name as their base attribute.

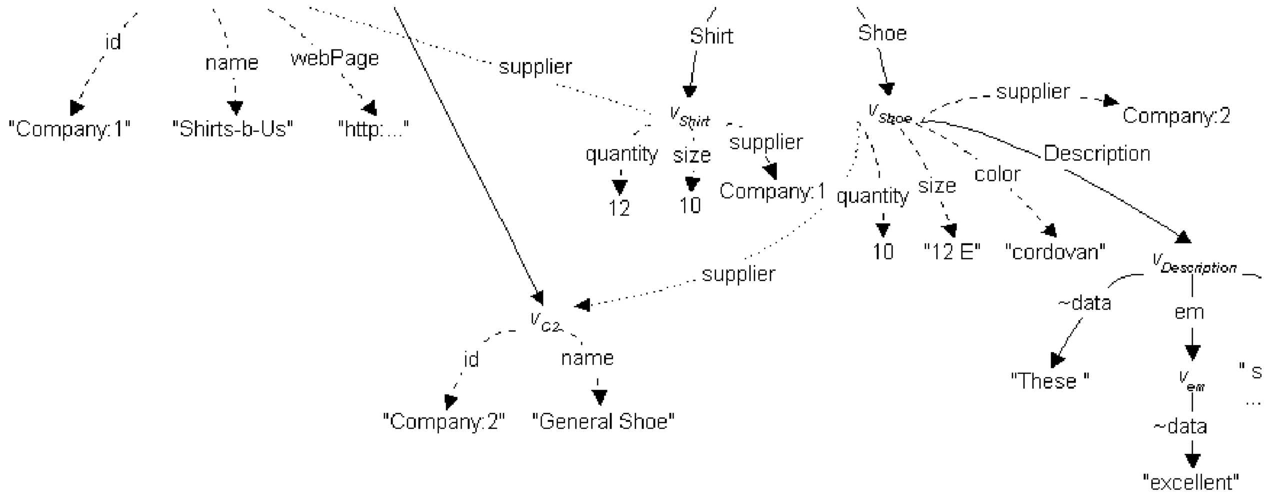
Complex Example

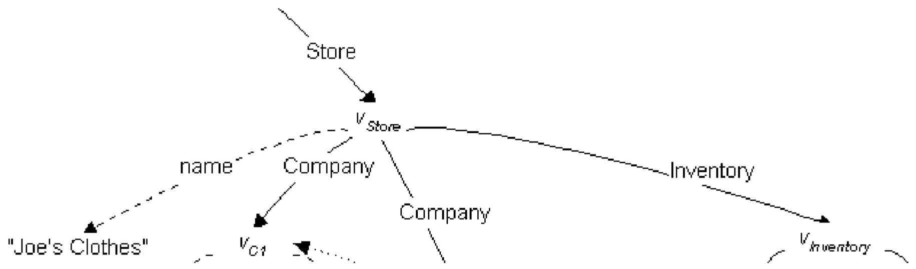
The following shows a more complex XML example and its data model graph. Element containment edges are represented as solid lines with arrows. Dashed lines with arrows denote attribute edges, dotted lines with arrows denote reference edges. We assume that there is enough schema information to identify `id` as an ID and `supplier` as an IDREF. The edges are mapped left to right according to their order.

```
<Store name="Joe's Clothes">

     <Company id="Company-1" name="Shirts-b-Us" webPage="http:www.shirts-b-
     us.com"/>
     <Company id="Company-2" name="General Shoe"/>
     <Inventory>
          <Shirt quantity="12" size="10" supplier="Company-1"/>
          <Shoe quantity="10" size="12 E" color="cordovan" supplier="Company-2">
               <Description>These <em>excellent</em> shoes have been made by our
               family of gnomes for seventy generations, using only the finest
               hides from virgin Naugas.</Description>
          </Shoe>
     </Inventory>

</Store>
```





Property Overview

The following table gives an overview over the simple properties on the vertices and the edges of the graph. The properties in *italics* are primitive operations while the operations in regular font are derived properties.

Component	**Operation**	**Result type**
Element vertex	*value*	unique identifier
	type	"element"
	gi (generic identifier)	name
	parent	element vertex
	referredby	list of element vertices: traversin back over reference edges
	childelements	list of element containment edge
	attributes	set of attribute edges
	referents	list of reference edges
Value vertex	*value*	value
	type	data type
Element containment edge	*name*	name
	parent	element vertex
	child	vertex
	type	E
	next	element containment edge
	previous	element containment edge
Attribute edge	*name*	name
	parent	element vertex
	child	value vertex
	type	A
	next	attribute edge
	previous	attribute edge
Reference edge	*parent*	refering element vertex
	refedges	list of edges that define the refer
	child	referred element vertex (derefere
	type	Rx where x indicates reference mechanism
	name	derived name
	next	reference edge
	previous	reference edge
Order (over a list of edges)	*e*	edge
	succ	successor of e
	pred	predecessor of e
	first	edge
	last	edge

	[n]	n-th edge
Name	*name*	unqualified name
	namespace	edge that provides namespace information
	qualifiedname	name with namespace prefix
	ns_uri	namespace URI

XML Algebra

Goals

The XML algebra should operate on a collection of XML documents in their data model representation. It should provide capabilities for selecting documents that meet given criteria and components of documents that meet given criteria. The algebra should also support the composition of XML documents from selected documents and their components.

The algebra should be minimal enough to provide an abstraction of the basic functionality, it should be composable into algebra "macros" that are more user-friendly and/or more efficient to implement (e.g., a relational join is a selection over a cartesian product).

It should be able to deal with the specific nature of XML, such as

- Graph structure
- Heterogeneity of types and graph structure
- Ownership vs reference
- Order (there should be an option to preserve it or not to care)

While preserving standard algebra virtues such as

- Composability
- Optimizability

As a basis for discussion, we propose the following operations. Every operation has an explicit result, and - as a side effect - transforms the underlying graph(s) into a new graph that will form the basis of the next operation.

The operations of the form o[f(x)](x:A) (including ϕ) are all specialized lambda-expressions that bind x to each element of A and apply f(x) according to the semantics of the operation o. This fact can be used to define the semantics of the operations more formally - a task that for now is left as an exercise to the reader.

Collections, Duplicates, Order and Unorder

The following operations will be composed to expressions that return collections of items (e.g., vertices, nodes or other collections). Most operations will preserve the input order in accordance to their semantics and not do any duplicate elimination unless required by the operation semantics.

In the following, we will use the term **sequence** to denote an item collection with a given order, that can be **partial** or **distinct** (in the case of no duplicates). The term **bag** denotes the unordered collection with duplicates, whereas the term **set** denotes the unordered collection without duplicates. Finally, we will use the generic term **collection** when any of the above is allowed.

Distinct

To eliminate duplicates in a collection, we provide the **distinct** operation

$$\delta(\text{expression})$$

δ takes an *expression* that evaluates to an collection as input and returns a collection containing the distinct members of the input collection i.e. it eliminates duplicates. The order of the original collection is maintained if present.

Unorder

In order to transform a sequence into a unordered collection (i.e., a set or bag), we provide the operation

$$\chi(\text{expression})$$

χ takes an *expression* that evaluates to an collection as input and returns the unordered collection. In order to guarantee a set, it would have to be combined with the **distinct** operation: $\delta(\chi(\text{expression}))$.

Sort

In order to reorder the sequence or to order any unordered collection, we provide the **sort** operation

$$\Sigma[\mathit{value\_expr}(x)](e{:}\mathit{expression})$$

Σ takes an *expression* that evaluates to a collection as input and returns a sequence ordered on the *value_expr*. *value_expr* needs to end in a value vertex. *value_expr* is followed for each member of the input collection and the result sequence is ordered according to the value of the value vertex.

Collation order and the value of the xml:lang may have an influence on the sort order. This is outside of the scope of the algebra.

Pick and flatten

In principle, most operators work over a collection of items. The **pick** operation performs an automatic conversion to a singleton collection (a sequence), if an operation results in a singleton but the subsequent operation expects a collection. However, a singleton set will not be automatically transformed into a singleton item when no collection is expected.

$$\mathit{pick}(\text{expression})$$

transforms an element of the collection specified by the expression into a singleton.

Some operations may create collections of collections. The **flatten** operation

$$\mathit{flatten}(\text{expression})$$

will flatten a collection of collections (of any nesting depth) into a flat collection.

Navigation

Path navigation allows us to start with a collection of vertices (or nodes) in the graph and to follow paths that are described by name and type. The basic operation of the algebra is **follow**:

ϕ[*edgetype*, *name*](*vertex-expression*)

This follows the edges with the specified *edgetype* and the given *name* that originate at the specified collection of vertices specified by the *vertex-expression*. Edge type is either E for element containment, A for attribute, R for reference, or a combination of the three (e.g., E | A means follow both element containment and attribute edges but not reference edges of the given name). The name is either a valid edge name or a set of names specified by a regular expression over the names (e.g., (*a*|*A*)*d*[*d*]*ress#* describes names that start with either *adress*, *Adress*, *address*, or *Address*). # is a wildcard that matches any name, similar to the XPath * operator.

vertex-expression describes a collection of vertices in the graph that form the source (i.e., they are the parents of the described edges). The result is the collection of all edges that are described by the follow operation. If *vertex-expression* is a sequence, the result order is determined first by the order of the *vertex-expression* and for each vertex in *vertex-expression* by the order of the outgoing edges. In the case of mixed edge types, this may lead to partial orderings because there is no order between the different edge types. The result collection contains unique edges unless *vertex-expression* contained duplicates.

Thus, in order to follow a path along A then B then C element containment edges starting at the element vertices denoted by the variable *var*, we would write:

ϕ[E, C](*child*(ϕ[E, B](*child*(ϕ[E, A](*var*))))),

where *child* is the edge property "child" described above, that given an edge (set of edges) returns its child vertex (children vertices).

The innermost function in the above expression starts with *var* and follows all element (E) edges called A from it. It then applies the *child* operation to the returned edges to get the sequence of vertices reached by these edges. The next outer operation follows all element edges called B starting from this set of vertices and so on.

Subsequently, we will also use the following abbreviated syntax similar to the abbreviated XPath syntax:

- *a*/b stands for: ϕ[E, b](*a*), where *a* stands for any vertex collection expression.
- *a*/@b stands for: ϕ[A, b](*a*), where *a* stands for any vertex collection expression.
- *a*/>b stands for: ϕ[R, b](*a*), where *a* stands for any vertex collection expression.

In all three cases, intermediate *child* calls will be implied. The result of a sequence of these path operations will, however, always be a collection of edges. With this syntax the above example may be written as: *var*/A/B/C.

The **inverse follow** operation ϕ_{inv} returns all the edges of the specified type and name that lead to the specified set of vertices.

Note that if we use concatenated *follow* and *inverse follow* along with *child* operations to find the descendants of some node collection, we lose the paths that got us to the result collection of vertices. In order to preserve intermediate edge traversals, paths can be bound to variables. E.g., if we wanted to preserve path traversals in the above example, we would write:

a := ϕ[E, A](*foo*); b := ϕ[E, B](*child*(a)); c := ϕ[E, C](*child*(b));

where the variables *a, b* and *c* preserve the collection of paths traversed at each stage.

Regular path expressions expressing alternative paths such as *root*/A/(B|C)/D where *root* denotes the root vertex, can be expressed as

$$\phi[E, D](\textit{child}(\phi[E, B|C](\textit{child}(\phi[E, A](\textit{root})))))$$

which is equivalent to

$$a := \phi[E, A](\textit{root});\ \phi[E, D](\textit{child}(\phi[E, B](\textit{child}(a)) \cup \phi[E, C](\textit{child}(a))))$$

where the union respects the order among the edges. Regular path expressions such as *root*/A/(B+C)/D where B+C denotes that two paths have to lead to the same objects are equivalent to

$$a := \phi[E, A](\textit{root});\ \phi[E, D](\textit{child}(\phi[E, B](\textit{child}(a)) \cap \phi[E, C](\textit{child}(a))))$$

where the intersection respects the order among the edges. Regular path expressions such as *root*/A/(B-C)/D where B-C denotes that the path has to lead to vertices that are reachable through B edges but not through C edges. This is equivalent to

$$a := \phi[E, A](\textit{root});\ \phi[E, D](\textit{child}(\phi[E, B](\textit{child}(a)) - \phi[E, C](\textit{child}(a))))$$

where - denotes the order-preserving difference operator.

Kleene star

The **Kleene star** operator $*$

$$*[f(x)](x{:}\textit{expression})$$

is a fixpoint operator that repeats the function *f* 0 to (potentially) infinite times starting with the initial seed given by the *expression*. After each iteration, the results of the function are added to the seed and the function is reapplied until we reach a fixpoint of the seed collection. Note that the resulting fixpoint will be ordered according to the order of the seed collection and the order implied by the recursively applied function. If the collection semantics of the seed collection is that of a bag, then the fixpoint is reached when no elements are added anymore, if it has set semantics, then the fixpoint is reached when the seed collection does not grow anymore.

The alternate syntax

$$*[f(x), n](x{:}\textit{expression})$$

denotes that the function has to be applied 0 to *n* times or until we have reached a fixpoint, whatever occurs first. It holds

$$*[f(x), 0](x{:}A) = A$$

For example, the Kleene-$*$ operator allows us to navigate among paths repeatedly. $*[\textit{pexpr}(x)](x{:}\textit{source})$ means repeat the path expression $\textit{pexpr}(x)$ 0 to infinite times starting with the initial collection of vertices given in *source* until we reach a fixpoint with respect to the collection of vertices reached. For example,

$$\phi[E, C](*[\textit{child}(\phi[E, \#](y)), 2](y{:}*[\textit{child}(\phi[E, B](\textit{child}(\phi[E, A](x))))](x{:}\textit{root})))$$

starts at the root vertex, follows the A/B path combination several times until it reaches a fix point, then follows all of the available edges two levels down, before following the final edge called C. In an abbreviated XPath syntax this could look like: *root*/(A/B)*/#*2/C.

Map

Often, the same complex expression should be applied over a collection of items (edges or vertices). The **map** operator

$$\mu[f(\mathrm{x})](x{:}expression)$$

allows one to apply the expression *f*(x) to each element of the collection generated by the *expression*. Unlike the Kleene-∗ operation, **map** does not include the input collection in the result collection, but only the result of the expression application. Assuming duplicate semantics, it holds:

$$\mu[f(\mathrm{x})](x{:}A) = *[f(x), 1](x{:}A) - A$$

In this paper, we implicitly employ the map operation when we apply a single vertex or edge property function over a collection. For example, *child*(*A*) is equivalent to μ[*child*(x)](*x*:*A*) if *A* denotes a collection.

For an example, see the last example of create edge.

Selection

A query algebra also needs a **selection** operator. The operator

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

allows us to express selection over a collection indicated by *expression*. The variable *e* is bound in turn to elements of the collection. If *condition*(*e*) evaluates *true* the corresponding element is added to the result collection. If it evaluates *false* (or if it returns UNKNOWN in case of a three-valued logic), *e* is not added to the result collection. The fact that the selection operator takes a collection as input and returns a collection can be used to compose selection operators.

Earlier we discussed properties of vertices and edges such as *value* and *type*. These can be used to construct selection conditions. Consider a simple case. The variable *var* contains a sequence of value vertices. We want to select the vertices where their value equals "3". This can be expressed as:

$$\sigma[value(v)=3](v{:}var)$$

The result will be either a sequence containing those elements in *var* that satisfy the condition, or the empty sequence if the condition is not satisfied.

In the same way, we can write selection operators on the type of the vertex on the GI of an element vertex and so on. Instead of equality in the above example, other standard comparison operators (such as <>, <, >, <=, >=), can be used. Boolean operators (*and, or, not*) can be used to create more complex conditions. Finally, other type specific operations such as string or numeric operations can also be used to form selection conditions.

Navigation can be used in selection conditions. For example, if we want to select all Customer names that have an order with today's date, we could write the following three expressions:

$$A := \phi[\mathrm{E, Customer}](*[child(\phi[\mathrm{E}, \#](x))](x{:}root));$$
$$B := \sigma[value(child(\phi[\mathrm{E, Date}](child(\phi[\mathrm{E, Order}](child(c)))))) = TODAY()](c{:}\mathrm{A});$$
$$C := child(\phi[\mathrm{E, Name}](child(\mathrm{B})));$$

A contains all Customer edges in the document. The $*$ operator reaches all vertices that are reachable from the root vertex (or vertices), and the follow operator then follows all existing Customer edges. The algebraic expression is complex and describes the process inefficiently. The query language will provide an operation with simpler syntax (such as the XPath expression "//Customer") that is easier on the eye and easier to optimize.

B contains all Customer edges that have at least one order with the requested date.

C is the final result: the requested Customer names.

Existential and Universal Quantification

Note that the operation that created the collection *B* was a selection over children, orders, of a parent, customer. We added the parent to the result set, i.e., selected the parent *if any of the children met the condition*. This, existential quantification, is the default. If we wanted *all* the orders to have today's date the above expression would have to be modified to:

$$\mathrm{B} := \sigma[\mathit{all}(\mathit{value}(\mathit{child}(\phi[\mathrm{E, Date}](\mathit{child}(\phi[\mathrm{E, Order}](\mathit{child}(c)))))) = \mathit{TODAY}())](c{:}\mathrm{A});$$

Existential quantification can be explicitly specified, if desired by using the *any* operator.

In the general case, with sets on both sides of the comparison operator we need to introduce iterator variables that let us iterate over the sets. $\mathit{any}(i, \mathit{pred}(i))$ and $\mathit{all}(i, \mathit{pred}(i))$ allow us to express conditions such as $\mathit{any}(i, \mathit{seqA}_i = \mathit{seqB}_i)$ and $\mathit{any}(i, \mathit{any}(j, \mathit{seqA}_i = \mathit{seqB}_j))$.

Joins

A query operation selects documents that meet stated criteria from a collection of documents. It may also extract components from selected documents and construct new documents from these components. In some cases, we may want to start with more than one source collection and perform the initial selection on documents from these collections that are related by some condition. This is called a **join**.

A join

$$(a{:}\mathit{expression})\otimes[\mathit{condition}(a, b)](b{:}\mathit{expression})$$

operates on the two sets of element vertices specified by the left and right expressions. These may, for example, be sets of vertices from two separate collections. The join is evaluated over the cartesian product of the element vertices. For each tuple (*a*,*b*), for which the join condition *condition*(*a*, *b*) evaluates *true*, it returns a single virtual reference edge from *a* to *b* that embodies the information in the join.

The reference edges introduced by the join are called *virtual* because they are not part of the document structure but are added as a part of the query process. Later we will see other examples of such virtual edges added as part of the query process.

Adding the *R* edges creates a single collection from the multiple source collections. It is this single collection that the rest of the query uses as the source.

Formally, the join can be defined as

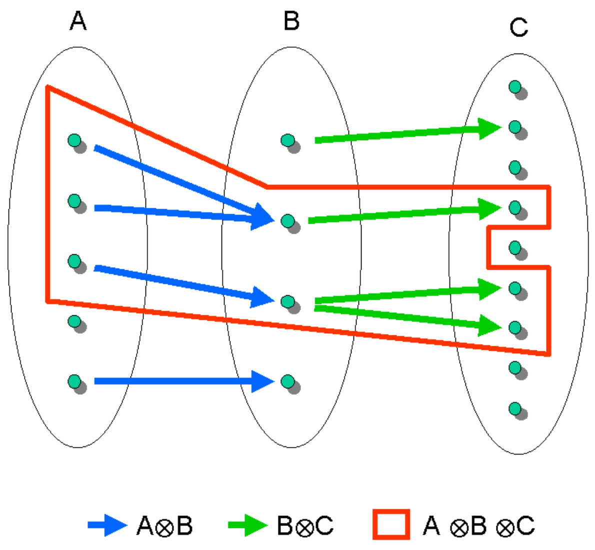
$$(a{:}A\subseteq V_{element})\otimes[\mathit{condition}(a, b)](b{:}B\subseteq V_{element})\equiv \{r \in R \mid \exists a \in A,\, b \in B{:}\mathit{condition}(a, b) \wedge r = \nu[R_{join}, \text{-}, b](a)\}$$

where $\nu[R_{join}, -, b](a)$ is the operation that creates the new join *R* edge from *a* to *b*. The resulting *R* edges preserve the order of the elements in the left join partner *A*.

N-way joins are expressed through join sequences such as

$$(a{:}expression)\otimes[condition(a, b)](b{:}expression)\otimes[condition(b, c)](c{:}expression).$$

The first join that is executed generates a collection of *R* edges and not a collection of element vertices. Therefore, the n-way join notation is a notational convenience. The result of such a join sequence is the union of the collection of *R* edges from *a* to *b* and the collection of *R* edges from *b* to *c* if and only if the first *b* is equal to the second *b*. If somebody would like to generate the collection of direct *R* edges from *a* to *c* only, an additional self-join needs to be performed. The following picture illustrates the result of a three-way join:



Formally, the n-way join can be expressed as

$$(a{:}A\subseteq V_{element})\otimes[condition(a, b)](b{:}B\subseteq V_{element})\otimes[condition(b, c)](c{:}C\subseteq V_{element})\otimes\ldots \equiv$$
$$\{r \in R \mid \exists a \in A, b \in B, c \in C, \ldots : condition(a, b) \wedge condition(b, c) \wedge \ldots \wedge (r = \nu[R_{join}, -, b](a) \vee r = \nu[R_{join}, -, c](b) \vee \ldots) \}$$

Note that this is different than

$$C := (a{:}A)\otimes[condition(a, b)](b{:}B);\ (c{:}C)\otimes[condition(c, d)](d{:}D)$$

because C is a collection of edges and not element vertices, which makes this join formulation invalid.

The n-way join is associative, i.e., it does not matter, in what order the joins are processed. On the other hand, commutativity is not preserved, because the introduced reference edges are directed and derive their order from the left join partner.

Several special cases are possible:

One-to-one Joins

Consider the case where we have a XML document that has the name, address and social security number of individuals and has as root *root1* and where we have another XML document with the social security number and investment portfolio for individuals and root *root2*. We would like to create a mailing list with the name, address and portfolio for each individual. To do this, we would write a join that matches the documents by social security number. Assume that the social security number is stored as an element in the first collection of documents and as an attribute in the second collection and are spelt differently.

The following join

(*a*: *child*(ϕ[E, Person](*root1*)))⊗[*child*(ϕ[E, ~*data*](*child*(ϕ[E, SocSecNo](*a*)))) = *child*(ϕ[A, SSN](*b*))](*b*: *child*(ϕ[E, Person](*root2*)))

creates *R* edges between the persons of the first and second document that have the same value and type for the social security number.

Note that the join

(*a*: *child*(ϕ[E, ~*data*](*child*(ϕ[E, SocSecNo](*child*(ϕ[E, Person](*root1*)))))))⊗[*value*(*a*) = *value*(*b*)](*b*: *child*(ϕ[A, SSN](*child*(ϕ[E, Person](*root2*)))))

would connect the vertices that represent the Social Security Number. This join is illegal, since join produced reference edges can only go from element to element vertex.

One-to-many Joins

Now, consider the case where we have a XML document that has information about employees identified by an employee number (rooted in *root1*) and another XML collection representing orders and their assigned employee (rooted in *root2*). We would like to match each employee with the orders initiated by him. To do this, we would write the following join that matches the documents by the employee number (we assume that no ID/IDREF reference is made explicit).

(*a*: *child*(ϕ[E, Employee](*root1*)))⊗[*child*(ϕ[A, empid](*a*)) = *child*(ϕ[A, employee](*b*))](*b*: *child*(ϕ[E, Order](*root2*)))

This join connects the two documents by creating multiple *R* edges from each employee vertex to the vertices representing orders initiated by him.

Many-to-many Joins

Now, let's suppose that there is a single, large XML document of car manufacturers and the various models they offer (rooted in *root1*) and that there is another single, large XML document of car dealers (rooted in *root2*). Each dealer element contains child elements with the car models he sells. With a join on model number:

(*a*: *child*(ϕ[E, Model](*child*(ϕ[E, Manufacturer](*root1*)))))⊗[*child*(ϕ[A, modelnumber](*a*)) = *child*(ϕ[A, modelNo](*b*))](*b*: *child*(ϕ[E, Dealer](*root2*)))

we create multiple *R* edges from each model in the manufacturer document to dealers who sell that model in the dealer document.

Additional join operations such as left outer joins can be expressed by composing joins and selections by means of collection operations such as unions.

Result Construction

The results of a query operation may just be the sub collection of the documents from the input set that satisfy the selection criteria. Often, however, the user wants to get back specified components of the selected documents. For example, from a set of books he may want only the titles and authors of the books. He may then want to process the returned values in various ways. A very important option is to create one or more XML documents from them. This can be done by using XSL transformations or by direct construction.

In this section we discuss how to specify what has to be returned and how to construct new XML documents from the document components that are returned. We will not discuss how to construct XML documents using XSL transformations on returned values.

Expose

The **expose** operation

$$\varepsilon[\mathit{edge1}(a), \mathit{edge2}(a), \ldots](a{:}\mathit{expression})$$

is a special map operation and exposes the edges *edge1, edge2,* etc. that originate at the set of vertices specified by the given expression. The exposed edges can be either already existing edges or edges that are newly generated using the edge creation operation explained below. The **expose** operation is object-preserving and returns a sub collection of the edges in the data model (after adding the virtual edges newly created by the edge creation operations). For examples of the expose operation and the create edge and create vertex operations defined below see the grouping operation.

The expose operation can be composed recursively in a manner similar to the follow operation. For example,

$$*[\varepsilon[\phi[E, \#](\mathit{child}(y))](y{:}x)](x{:}\mathit{root})$$

exposes all the *E* edges from the root recursively as far as possible. In effect, it returns the whole document tree without attributes and references.

Analogously, we can define a hide or "except" operator or an argument to expose that exposes all sub-edges *except* the specified ones. This would allow us to be able to deal with open content where we might not know a priori what is there but only what we do not want to provide.

Create Vertex

This operation generates a simple vertex in the graph. Note that it cannot stand alone in a valid expression but always needs to be an argument to an edge creation. The syntax is

$$\nu[\mathit{type}](\mathit{value})$$

where *type* is the vertex type and *value* the vertex value if the type is concrete. For example, ν[integer](2*21) creates a new integer vertex with the value 42. When creating an element vertex using ν[element](), no value needs to be given, since the object id will be created automatically (any given value will be ignored). Note that the element's GI is the name of the edge and will be given when creating the element containment edge.

Copying a vertex *v* can be done with the statement $\nu[\mathit{type}(v)](\mathit{value}(v))$.

Create Edge

Edges are created with

$$\nu[\text{edgetype}, \mathit{name}, \mathit{child}](\mathit{parent})$$

where the *edgetype* is either A for an attribute edge or E for an element edge. *R* edges are created implicitly by creating the correct attribute and/or element containment edges and the correctly typed vertices. Virtual *R* edges are generated by joins. *name* gives either the attribute name or the element's GI, whereas *child* is the child vertex (in the case of an attribute edge of a concrete type) and *parent* is the parent element vertex of the edge. Note that the names ~*data*, ~*comment* and ~*PI* are reserved to create the special containment edges.

The order in which edges with the same parent are created gives the default order in the order relation *O*. Newly created edges will be appended at the end. In order to change an edge's position, an optional position parameter could be added

For example,

a := ν[E, "MyRoot", ν[element]()](*null*)

creates a new root element and assigns it to *a*.

ν[E, "~*data*", ν[string]("This is a test")](*b*)

adds the string "This is a test" as content to the element *b*.

ν[A, "creation", ν[date](*TODAY*())](*a*)

adds a new attribute named creation with today's date to the newly created MyRoot element.

For children with concrete types, we can also use the short form ν[edge type, *name*, *value*](*parent*) if the type is clear from the value (e.g., ν[A, "author", "Jane Doe"](*a*)).

The operation will make sure that no ill-formed XML can be generated. For example, in order to copy an edge, the vertex that it points to needs to be copied as well.

The following operation using the **map** and the **Kleene star** clones the complete subgraph of that starts at an element *foo*:

μ[ν[*type*(*e*), *name*(*e*), ν[*type*(*child*(*e*))](*value*(*child*(*e*)))]](*e*: ∗[φ[E | A, #](*child*(*x*))](*x*: φ[E | A, #](*foo*)))

Return

Often, queries should make copies of the result. Instead of using the **map** and create operations, the **return** operation

ρ[*edge1*(*a*), *edge2*(*a*), ...](*a*:*expression*)

simplifies this frequent operation. **return**, like the **expose** operation, returns the edges *edge1, edge2*, etc. that originate at the set of vertices specified by the given expression. The returned edges can either be already existing edges or edges that are newly generated using the edge creation operation explained above. Unlike **expose**, the **return** operation is object-generating and generate *copies* of the vertices and edges into a new graph.

Grouping

Grouping operations can be used to create element and attribute vertices in the result that aggregate or summarize information from a group of similar vertices or edges. Several styles of grouping operations are possible. For example, the substructure:

```
<Customer>
     <Order oid="o1" price="4"/>
     <Order oid="o2" price="6"/>
</Customer>
```

may be grouped and aggregated as:

```
<Customer avgprice="5" />
```

or

```
<Customer avgprice="5">

     <Order oid="o1"/>
     <Order oid="o2"/>
</Customer>
```

or

```
<Customer>

     <avgprice>5</avgprice>
</Customer>
```

The **group-by** operator

$$\gamma[\mathit{grouping\_expr}(x)](x{:}\mathit{expression})$$

creates groups from a collection of items bound to *x* based upon the values of the *grouping_expr* i.e. members of each group have the same value of the *grouping_expr*. Thus, groups form disjoint subcollections of the original collection generated by the expression bound to *x*.

For example, one may want to create groups of employees grouped by department number from a list of employees. The following would create groups of employee vertices where each members of each group would have the same department number.

EmpList:= *child*(ϕ[E, Employees](*root*))
EmpGroups:= γ[*value*(*child*(ϕ[A, Dept_Number](*a*)))](*a:EmpList*)

The default ordering of the created groups is by the order of the first element in the original collection. The sort operation can be used to reorder the groups. Its order expression *value_expr* may be a scalar function of properties of the members of the group such as *avg* or it could be the grouping expression itself. For example, to order the employee groups by department number one would write:

EmpGroups:= Σ[*value*(*child*(ϕ[A, Dept_Number](*x*)))](*x*: γ[*value*(*child*(ϕ[A, Dept_Number](*a*)))](*a*:*EmpList*))

Together with the other operators in the algebra and aggregation functions such as *sum*, *avg* etc., we can specify the aggregations presented above. For example, given the XML document fragment

```
<Book bid="b1" authors="a1 a2"/>
<Book bid="b2" authors="a3 a2"/>
<Author aid="a1" name="A"/>
<Author aid="a2" name="A"/>
<Author aid="a3" name="C"/>
```

where `authors` is of type IDREFS and `aid` of type ID, the algebra expressions

A:= *child*(φ[E, Book](*root*))
Agroup:= γ[*child*(*a*)](*a*: φ[R, authors](A))
B:= μ[ε[ν[E, "Aut", *x*](*null*)
    , φ[A, name](*x*)
    , ν[A, "no_of_books", ν[integer](*count*(*ga*))](*x*)
    ](*x*: δ(*child*(*ga*)))
  ](*ga*:Agroup)

group the author element edges by their child vertices (Agroup) and expose their name (using the already existing edges) and the aggregation over their number of books. Assuming that r_{a1} and r_{a2} denote the two reference edges implied by the authors attribute of the first book and r_{a3} and r_{a4} denote the two reference edges implied by the authors attribute of the second book, Agroup is $\{\{r_{a1}\}, \{r_{a2}, r_{a4}\}, \{r_{a3}\}\}$. The map operation then applies the creation and projection of the new "Aut" elements, their name and no_of_books attributes. The resulting XML fragment referenced by B looks like:

```
<Aut name="A" no_of_books="1"/>
<Aut name="A" no_of_books="2"/>
<Aut name="C" no_of_books="1"/>
```

The following algebra expression groups according to the author names and provides the aggregation of the number of books written by authors with the same name as subelement together with the ids of the corresponding authors. It generates all new copies for the element type vertices.

A:= *child*(φ[E, Book](*root*))
Agroup:= γ[*value*(*child*(φ[A, name](*child*(*a*))))](*a*: φ[R, authors](A))
B:= μ[ε[ν[E, "Aut", *x*](*null*)
    , ν[A, "name", *child*(φ[A, name](*child*(*pick*(*ga*))))](*x*)
    , μ[ν[E, "aid", *parent*(ν[E, ~*data*, ν[integer](*value*(*child*(φ[A, aid](*y*))))](ν[element]()))](*x*)](*y*: δ(*child*(*ga*)))
    , ν[E, "no_of_books", *parent*(ν[E, ~*data*, ν[integer](*count*(*ga*))](ν[element]()))](*x*)
    ](*x*: ν[element]())
  ](*ga*:Agroup)

generates

```
<Aut name="A">

      <aid>a1</aid>
      <aid>a2</aid>
      <no_of_books>3</no_of_books>
</Aut>
<Aut name="C">
      <aid>a3</aid>
      <no_of_books>1</no_of_books>
</Aut>
```

Future Work

The algebra described above can be made yet more powerful by the addition of a few advanced functions such as the ones mentioned below and by adding manipulation functions (update etc.).

Skolem functions:

$$\nu_{Skolem}[idexpr, type](value)$$

generates a new vertex of the given *type* with the given *value* and the object id specified with *idexpr*. Similarly, the function

$$\nu_{Skolem}[idexpr, type, name, child](parent)$$

generates a new edge. If a vertex or edge with the given id already exists, the Skolem functions return the already existing vertex or edge instead of generating a new one. This functionality can be used in result construction to fuse objects based on the *idexpr*. For example, assume, that the collections *C1* and *C2* contain two different customer representations. *C1* customers are identified by a subelement called Id, *C2* customers by an attribute cid. For the sake of simplicity we assume that the same value in either of these fields identify the same customer. The following application of the Skolem function generates a copy for each customer and adds the id as customerid attribute. The Skolem functions guarantee that only one node and edge is generated for each customer with the same id.

$\mu[\nu_{Skolem}[value(child(x/Id)), \mathrm{A}, customerid, value(child(x/Id))](\nu_{Skolem}[value(child(x/Id)), \mathrm{element}]())](x{:}C1) \cup \mu[\nu_{Skolem}[value(child(x/@cid)), \mathrm{A}, customerid, value(child(x/@cid))](\nu_{Skolem}[value(child(x/@cid)), \mathrm{element}]())](x{:}C2)$

Bound function:

$$bound[set\_of\_bvar2](bvar1)$$

returns all instances that can be bound to the bound variable *bvar1*, such that they contain the set of bound variables *set_of_bvar2*. This can be used to invert hierarchy in construction. Example: years contains months contains regions contains sales. We have bound the years to y, months to m, regions to r and sales to s. Now the result should aggregate the sales by regions and by year. Here we would generate a new region object using the Skolem function (id on region name), and would use *bound* to get to any year y that binds to the regions with the same region name to get to their sales figures that are then aggregated.

References

[Infoset]
The XML Information Set Specification.
[Namespace]
The XML Namespace Specification.
[XPath]
The XPath Specification.