Uppsala Student Thesis
Computing Science No. 268
2004-01-30
ISSN 1100-1836

Translating XQuery expressions to Functional Queries in a Mediator Database System

A student project paper by
Tobias Hilka

Advisor and Supervisor
Tore Risch

January 2004

Uppsala Database Laboratory
Uppsala University
P.O. Box 513
S-751 20 Uppsala
Sweden

Contents

1 Introduction

This report on my student project will discuss some approaches of translating XQuery expressions into functional queries in a mediator database system. The mediator database system used is AMOS II (Active Mediator Object System)[1][2]. This system uses a functional data model and is based on the query language AmosQL. This language is relationally complete object-oriented and is similar to the object-oriented parts of SQL-99.

An Amos II database system can integrate data of different type into its own object-oriented database using wrappers. The wrappers process data from different external sources such as ODBC based relational databases, CAD systems or internet search engines. This results in a common data model and a query language for heterogeneous data.

AMOS II can not only be used as a stand-alone database system but also interoperate with many other independent AMOS II system over a communication network such as the internet. Applications can access data stored in some AMOS II database using other AMOS II databases called mediators. These mediators combine the underlying data sources in the needed way to offer a high-level abstraction. This greatly simplifies accessing heterogeneous data sources at the application level. The mediator system themselves can also manage data of its own.

The data stored in an AMOS II database resides in main memory until it is stored explicitly to hard disk. There are two kinds of interfaces between AMOS II and the programming language Java called the callin and the callout interfaces[3]. Using the callout interface, the programmer can define foreign AMOS II functions in Java. These functions can freely be used in database queries. To manipulate the database, foreign functions written in Java use the callin interface by calling AmosQL statements or functions.

The purpose of the project presented here was to find a way to translate XQuery expressions into queries expressed in AmosQL. Since XQuery is a very powerful language, not all constructions that are possible with XQuery can be translated to AmosQL. I used the well known XMLSchema based Waterloo benchmark for XML databases[4] and the queries used to generate workload.

1.1 XML

XML stands for Extensible Markup Language and was standardized by the W3C[5]. It is designed to structure, store and send information. The tags used in XML are not predefined like in HTML and can be defined using a Document Type Definition (DTD) or an XML Schema[6]. Because of that, XML is extensible. With this standard it is possible to exchange data (e.g. over the internet) between different systems that are not able to communicate otherwise. Since its introduction by the W3C, XML has gained attention throughout the industry and is very widely used in software development.

1.2 XPath

As increasing amounts of information are stored, exchanged, and presented using the XML format, the ability to intelligently query XML data sources becomes increasingly important.

The primary purpose of XPath[7] is to address parts of an XML document. It operates on its abstract, logical structure. XPath gets its name from its use of a path notation as in URLs for navigating through the hierarchical structure of an XML document. In addition to its use for addressing, XPath is also designed so that it has a natural subset that can be used for matching (testing whether or not a node matches a pattern).

The primary syntactic construct in XPath is the expression. One important kind of expression is a location path which selects a set of nodes relative to the context node. There are two kinds of location path: relative location paths and absolute location paths. A relative location path consists of one or more location steps separated by '/'. The steps are composed from left to right. Each step selects a set of nodes relative to the context node. Every node in this set will be used as context node for the following step. An absolute location path consists of '/' optionally followed by a relative location path. A '/' by itself selects the root node of a document which contains as a child the element node for the document, which will be used as context node for the following step. A location path itself consists of the following parts: an axis, specifying the tree relationship between the nodes, a node test, specifying the node type and name of the nodes to be selected and optional predicates, refining the nodes selected by the location step. For example, in "`child::paragraph[position()=last()]`", `child` is the name of the axis, `paragraph` is the node test and `[position()=last()]` is a predicate selecting the last paragraph child of the context node. The example uses the unabbreviated syntax. There are a number of syntactic abbreviations which make it possible to write the example as `paragraph[last()]`.

A fully working extension to the AMOS II system, evaluating XPath expressions was done in a previous project called AXE[]. The AXE extension is used in this current project for the necessary evaluation of XPath expressions.

1.3 XQuery

XQuery Version 1.0 [8] is an extension to the XPath language. Therefore they share the same data model. Any expression that is syntactically valid and executes successfully in both languages will return the same result. XQuery is designed to be a language in which queries are concise and easily understood. It is also flexible enough to query a broad spectrum of XML information sources, including both databases and documents.

The basic building block of XQuery is the expression. There are several kinds of expressions provided by XQuery. This work focused on the translation of the so

called FLWOR-exrpressions as presented in the queries of the Waterloo benchmark. These expressions support iteration and binding of variables to intermediate results. This kind of expression is often useful for computing joins between two or more documents and for restructuring. The name FLWOR (pronounced "flower") consists of the first letter of the keywords `for`, `let`, `where`, `order by` and `return`.

The purpose of the `for` and `let` clauses is to generate a sequence of bound variables, the tuple stream. The simplest example of a `for` clause contains one variable and an associated expression. It evaluates the expression and iterates over the items in the resulting sequence, binding the variable to each item in turn.
A `for` clause may also contain multiple variables, each with an associated expression. In this case, the `for` clause iterates each variable over the items that result from evaluating its expression. The resulting tuple stream contains one tuple for each combination of values in the Cartesian product of the sequences resulting from evaluating the given expressions.

A `let` clause may also contain one or more variables, each with an associated expression. The difference compared to a `for` clause is that a `let` clause binds each variable to the result of its associated expression, without iteration. The variable bindings generated by let clauses are added to the binding tuples generated by the `for` clauses. If there are no `for` clauses, the `let` clauses generate one tuple containing all the variable bindings.

A simple example will illustrate that:
- Query 1:

```
for $s in (<x/>, <y/>, <z/>)
let $i := (<a/>, <b/>)
return <out>{$s}{$i}</out>
```

Output:

```
<out> <x/> <a/> <b/> </out>
<out> <y/> <a/> <b/> </out>
<out> <z/> <a/> <b/> </out>
```

- Query 2:

```
for $i in (<a/>, <b/>)
for $s in (<x/>, <y/>, <z/>)
return <out>{$s}{$i}</out>
```

Output:

```
<out> <x/> <a/> </out>
<out> <y/> <a/> </out>
<out> <z/> <a/> </out>
<out> <x/> <b/> </out>
<out> <y/> <b/> </out>
<out> <z/> <b/> </out>
```

- Query 3:

```
let $i := (<a/>, <b/>)
let $s := (<x/>, <y/>, <z/>)
return <out>{$i}{$s}</out>
```

Output:

```
<out> <x/> <y/> <z/> <a/> <b/> </out>
```

The optional where clause serves as a filter for the tuples generated by the `for` and `let` clauses. It is evaluated once for each tuple. If the effective boolean value of the where-expression is true, the tuple is used in the execution of the return clause. If it is false, the tuple is discarded.

The `return` clause is evaluated once for each tuple in the stream and the results of these evaluations are concatenated to form the result of the FLWOR expression.

The `order by` clause contains one or more ordering specifications. For each tuple in the stream, these specifications are evaluated. In the current version of this project, only ordering specifications that are results of the FLWOR expression can be used as ordering key.

2 First Approach

The first approach to the translation of XQuery to AmosQL was to use a system developed at UBDL that uses XML Schema to generate a database schema for AMOS II [9].
The basic idea behind this project was to parse a given XML Schema definition and translate it into the corresponding schema definitions in AMOS II. In other words, to dynamically generate a file containing the type and function definitions corresponding to the imported XML Schema. After the XML Schema specification is imported to AMOS II it is possible to populate the AMOS II main-memory database with any XML document that is described by this schema. After the schema is imported to AMOS II and the database is populated it is possible to state arbitrary queries over the imported data using AmosQL.
Here is an example of a XML Schema file describing a note:

```
<schema>
  <element name="note">
    <complexType>
      <sequence>
        <element name="to" type="string"/>
        <element name="from" type="string"/>
        <element name="heading" type="string"/>
        <complexType>
          <element name="body" type="string"/>
          <attribute name="lang" type="string"/>
        </complexType>
      </sequence>
    </complexType>
  </element>
</schema>
```

The corresponding AMOS II schema would look like this:

```
create Type XS_note;
create Type XS_to;
create Type XS_from;
create Type XS_heading;
create Type XS_body;

create function XS_to(XS_note)->XS_to as stored;
create function XS_from(XS_note)->XS_from as stored;
create function XS_heading(XS_note)->XS_heading as stored;
create function XS_body(XS_note)->XS_body as stored;
create function XS_to(XS_to)->charstring as stored;
create function XS_from(XS_from)->charstring as stored;
create function XS_heading(XS_heading)->charstring as stored;
create function XS_body(XS_body)->charstring as stored;
create function XS_lang(XS_body)->charstring as stored;
```

As you can see, all <element> tags are translated in AMOS II types. For every element there is a function inserted which returns the element of a given the parent node. For all <element> and <attribute> tags of type string in XML Schema a function returning the corresponding string of the element is returned.

A simple example of a possible translation of an XQuery query to an AmosQL Query using this database schema is

TC/MD_Q1:
Return the title of the article that has matching id attribute value (1).

XQuery:
```
for $art in input()/article[@id="1"]
return
   $art/prolog/title
```

AmosQL:
```
select XS_title(XS_title(XS_prolog(a)))
from   XS_article a
where  XS_id(a)=1;
```

The translation is possible in simple cases like the example above. An example using some more advanced features of the XQuery language is the following:

TC/MD_Q9:
Return all author names (several consecutive elements unknown) of the article with matching id attribute value (2).

```
for $art in input()/article[@id="2"]
return
   $art//author[position()=last()]/name
```

The `[position()=last()]` predicate was added to make the problems of the translation more obvious.

First of all, it is very hard to translate queries that use the "`descendant-or-self::node()`" axis. In abbreviated syntax this expression would be"`//`" like in the example above. The meaning of this construct is: select all nodes named `author` that are descendants of the document root node `article[@id="2"]`. The predicate is evaluated after the selection (see later). Since there is only the parent or the child of a node known, constructs like this need a `descendant(object)` function. Such a function could be constructed by a generic function which looks up all corresponding functions to this object in the schema and creates some kind of descendant function.

Another problem of this approach can not be solved. It concerns the non-document-preserving storage of the XML document in the database. There is no way to translate queries that refer to the structure of the document. Predicates like "`[postion()=last()]`" of the example can not be translated. Once stored in the database, the order of the elements is lost.

The main interest of the translation of XML Schema into the corresponding AMOS II schema was data-centric which means, only the content of an XML document is important. This type of documents are mostly intended to be processed by software. One example for this kind of documents would be air travel information. Document-centric documents (or text-centric as in the waterloo benchmark) in contrast are mostly for human consumption. The structure

of the document and particularly the order of the elements are of great interest. Documents-centric documents are for example web pages or user manuals.

Data centric queries are relatively easy to express over the translated XMLSchema using AmosQL. However, XQuery is mainly document centric and the queries are specified as paths through the document. Because of this the method of querying the translated data centric XMLSchema document is not so straight forward with XQuery. Therefore we decided to base the query translator on the document centric AXE system representation that represent the document structure explicitly and uses XPath to navigate the document structure.

3 Translation

Since the first approach turned out to be not capable to satisfy the requirements of document-centric query expressions, a new approach was taken. This new approach uses the AXE system developed at the UBDL.

As mentioned before, the AXE system is a general wrapper for schema-free XML documents. It also includes an implementation of XPath expressions. To represent the XML structure in AMOS II, the system used a DOM-like database schema. DOM stands for Document Object Model. It is a specification to describe the structure of documents and the way to access and manipulate documents. DOM uses a tree like object graph to model the documents that it represents. The mapping to the AMOS II database schema is done in a rather straightforward way.

With the help of the document preserving storage in the database and the implementation of the XPath expressions this system is suitable to serve as a platform for translating XQuery expressions to AmosQL.

The current version of the translator only works with AXE Version 12.

3.1 AmosQL and XQuery

The most flexible way to specify queries in AmosQL is to use the select statement. The simplified syntax of a select statement that is used for this translation is

```
select-stmt ::= "select" ["distinct"] expr-commalist
                [from-clause]
                [where-clause]
from-clause ::= "from" variable-declaration-commalist

variable-declaration ::= type-name variable-name |
        "bag of" type-name variable-name

where-clause ::= "where" predicate-expression
```

The [into-clause] of the original syntax is skipped because for the translation there is no use to have temporary variables in the AMOS II system.
Even more simplified, the select statement has the following format:

```
select result
from   type extents
where  condition
```

The general semantic of an AmosQL query is

- form the Cartesian product of the type extents
- restrict it by the condition
- for each possible variable binding to tuple elements in the restricted cartesian product, evaluate the result expressions
- result containing NIL are not included in the result set

Since it would be very inefficient to execute the query like this, the select statement passes a query optimization before being executed.
The partial grammar of the XQuery FLWOR expressions that can be translated looks like this:

```
FLOWRExpr ::= (ForClause | LetClause)+ WhereClause? OrderByClause?
"return" ExprReturn
```

Translating the whole scope of possible constructs of FLWOR expressions is far beyond the scope of this work, therefore some restrictions were made.

```
ForClause ::= "for" "$" VarName "in" PathExpr ("," "$" VarName
   "in" PathExpr)*
LetClause ::= "let" "$" VarName ":=" PathExpr ("," "$" VarName
   ":=" PathExpr)*
WhereClause ::= "where" Expr
OrderByClause ::= "order" "by" OrderSpec
PathExpr ::= ("/" RelativePathExpr?) | ("//" RelativePathExpr) |
   RelativePathExpr
ExprReturn ::= ((count(VarName)) | (VarName PathExpr)) | ("{"
   (count(VarName)) | (VarName PathExpr) "}")*
OrderSpec ::= VarName PathExpr
```

The OrderSpec part has to be one of the elements in ExprReturn since the sorting abilities of AMOS II are very restricted.

3.2 The Parts of FLWOR expressions

To illustrate the way the XQuery queries are translated into AmosQL some simple examples from the Waterloo benchmark are used. In this benchmark the expression text-centric is used for the expression document-centric documents that was used so far. The queries are split into four groups:

- **T**ext-**C**entric/**S**ingle-**D**ocument (TC/SD),
- **T**ext-**C**entric/**M**ultiple-**D**ocument (TC/MD)
- **D**ata-**C**entric/**S**ingle-**D**ocument (DC/SD)
- **D**ata-**C**entric/**M**ultiple-**D**ocument (DC/MD)

As an orientation for the reader, the group and the number of the queries are given for every example.

The translated queries were generated by a software tool which was implemented during this project. Some of the queries may have been written in an easier way by the user. Since translation rules are used for generating the AmosQL queries they may look a little complicated. The translation rules will be explained in chapter 4.

The path expressions used in the waterloo benchmark usually start with a function called `input()`. This is the unnamed input set of the database and is skipped in the translation of the expressions.

3.2.1 Simple Query using `for`

TC/MD_Q1:

Return the title of the article that has matching id attribute value (1).

XQuery:
```
for $art in input()/article[@id=”1”]
return
  $art/prolog/title
```

AmosQL:
```
select r0
from node f0, node xf0, node r0
where f0=evaluate(xf0, "/article[@id='1']")
and r0=evaluate(f0, "/prolog/title");
```

As you can see, for every `for` statement a new type extent is inserted in the `from`-clause. The same is done for every `return` statement. Furthermore, the name of the variable from the `return` statement is inserted into the result clause. To get the requested nodes, the

```
evaluate(node subtree, charstring expression) -> bag of node b
```

function of the AXE system is used. It takes a node (the root node for this evaluation) and an expression as arguments, evaluates the expression and returns a bag of nodes matching this expression. As you can see, in the statement for the result variable, `f0` (each node returned by the evaluate function of the `for` clause) is used as root node. Therefore, only the relative path from this node to the result node is used, just as it is expressed in the XQuery statement.

The query above can also be written in a much more simple:
```
select r
from node x, node r
where r=evaluate(x, "/article[@id='1']/prolog/title");
```
As mentioned before, the reason for the more complicated queries are the translation rules used to generate the queries.

3.2.2 `Where` Clause

An example using a simple where clause is

TC/MD_Q2:
Find the title of the article(s) authored by (Ben Yang).

XQuery:
```
for $prolog in input()/article/prolog
where $prolog/authors/author/name="Ben Yang"
return
  $prolog/title
```

AmosQL:
```
select r0
from node f0, node xf0, node tmp_f0, node r0
where f0=evaluate(xf0, "/article/prolog")
and tmp_f0=evaluate(f0, "/.[authors/author/name='Ben Yang']")
and r0=evaluate(tmp_f0, "/title");
```

This simple `where` clause is directly translated into a call to the `evaluate` function using `f0`, the variable it refers to, as root node. It uses the abbreviated syntax of XPath "`/.[predicate]`". Because of the slash, the evaluation starts with the root node (here all `prologs`). The "`.`" takes the current node and the condition in brackets is evaluated to restrict the current node. Since `f0` contains all nodes named "`prolog`" which have the parent node "`article`", this statement selects only the "`prolog`" nodes, which have an author named "Ben Yang" following the given path. After that, the "`title`" of the "`prolog`"(respectively the "`article`") is extracted.
There are some more complicated ways of expressing a where clause.

empty(pathExpr)
A query that illustrates the empty(pathExpr) function is

TC/MD_Q14:
List article title that does not have genre element.

XQuery:
```
for $a in input()/article/prolog
where empty ($a/genre)
return
   <NoGenre>
      {$a/title}
   </NoGenre>
```

AmosQL:
```
select r0
from node f0, node xf0, node r0
where f0=evaluate(xf0, "/article/prolog")
and notany(evaluate(f0, "/genre"))
and r0=evaluate(f0, "/title");
```

The `empty` expression of XQuery is directly translated into a call to the `evaluate` function, using the variable of the `for` statement as root node and passing the returned nodes to the AmosQL function `notany`. This function returns true if the bag passed as argument is empty.

contains(pathExpr, String)
The contains function is illustrated using the following example.

TC/MD_Q17:
Return the titles of articles which contain a certain word ("hockey").

XQuery:
```
for $a in input()/article
where contains ($a//p, "hockey")
return
   $a/prolog/title
```

AmosQL:
```
select r0
from node f0, node xf0, node r0
where f0=evaluate(xf0, "/article")
```

```
and some(select x from node x, text t where x=evaluate(f0, "//p")
  and like(nodeValue(t), "*hockey*") and parentNode(t)=x)
and r0=evaluate(f0, "/prolog/title");
```

The `contains` function of XQuery is translated into a rather complicated expression in AmosQL. The inner query within the `some` function uses the `evaluate` function to get all the nodes named `p` which are descendants of the root node `f0`. Then, an element of type `text` is used to compare the string, given by the `contains` function to the value of this element. The function

```
like(Charstring string, Charstring pattern) -> Boolean b
```

of AmosQL is used for the comparison. The `text` element is a child node of a `p` node. If at least one of these elements exist, the `some(bag of node) -> boolean` function of AmosQL will return true and the tuple will be used to evaluate the result expression.

Two more complicated expressions of XQuery are `some` and `every`. These expressions require special treatment.

some …
One example showing the use of the `some` expression is

TC/MD_Q6:
Find titles of articles where both keywords ("the" and "hockey") are mentioned in the same paragraph of abstracts.

XQuery:

```
for $a in input()/article
where some $b in $a/body/abstract/p satisfies
  (contains($b, "the") and contains($b, "hockey"))
return
  $a/prolog/title
```

AmosQL:

```
select r0
from node f0, node xf0, node w, node r0
where f0=evaluate(xf0, "/article")
and some(select x from node x, text t where x=evaluate(w, "/.")
  and like(nodeValue(t), "*the*") and parentNode(t)=x)
and some(select x from node x, text t where x=evaluate(w, "/.")
  and like(nodeValue(t), "*hockey*") and parentNode(t)=x)
and w=evaluate(f0, "/body/abstract/p")
and r0=evaluate(f0, "/prolog/title");
```

Inner queries are used to evaluate the XQuery `some` expression. One for each statement of the `some` expression. The first call to the evaluate function in each inner query selects all elements, that are addressed by the path of the new variable $b plus additional path given for each statement (in this case, no additional path is given, so "." is inserted). The `contains` function is translated with the help of a `text` element which is child of a `node x` and the `like` function of AmosQL.

every…
An example will illustrate the `every` expression.

TC/MD_Q7:
Find titles of articles where a keyword ("hockey") is mentioned in every paragraph of abstract.

XQuery:
```
for $a in input()/article
where every $b in $a/body/abstract/p satisfies
  contains($b, "hockey")
return
  $a/prolog/title
```

AmosQL:
```
select r0
from node f0, node xf0, node r0
where f0=evaluate(xf0, "/article")
and count(select x from node x, text t where x=evaluate(f0,
  "/body/abstract/p")and like(nodeValue(t), "*hockey*") and
  parentNode(t)=x)
=count(select x from node x where x=evaluate(f0,
  "/body/abstract/p"))
and r0=evaluate(f0, "/prolog/title");
```

The `every` function of XQuery is translated to AmosQL using two calls of the count function. The idea behind this is to have one inner query which has no restrictions and selects all the nodes of a path (in this case, the second call to count) without restriction. A second inner query(in this case the first one) selects all nodes of a path with the restriction given by the every statement. If the `count` of both queries is the same, *every* node of the given path satisfies the condition.

3.2.3 `Order By` Clause

The following example demonstrates the translation of the `order by` clause.

TC/MD_Q11:
List the titles of articles that have a matching country element type (Canada), sorted by date.

XQuery:
```
for $a in input()/article/prolog
where $a/dateline/country="Canada"
order by $a/dateline/country
return
  <Output>
    {$a/title}
    {$a/dateline/country}
  </Output>
```

AmosQL:
```
sort((select r0, r1
  from node f0, node xf0, node tmp_f0, node r0, node r1
  where f0=evaluate(xf0, "/article/prolog")
  and tmp_f0=evaluate(f0, "/.[dateline/country='Canada']")
  and r0=evaluate(tmp_f0, "/title")
  and r1=evaluate(tmp_f0, "/dateline/date")
```

```
), 2, 'inc');
```
In this example there are two results, therefore two type extents are inserted. The order by clause is not directly translated. As mentioned before, only elements that appear as results of the query can be used as sorting key. The reason for this is the sorting function of AMOS II:

```
sort(bag b, integer pos, charstring order) -> vector v
```

This function takes a `bag`, containing the tuples to sort, an `integer`, giving the position in the tuple by which to sort (starting with 1), and a `string`, either 'dec' or 'inc' giving the order of the sorting.

3.2.4 special return clause

The normal way of the translation of the `return` clause is obvious looking at the examples in this chapter. The `return` clause is translated into a call of `evaluate… .`
If the only result value is the count of a bag of nodes, the translation is different. Looking at a simple example will demonstrate it. The query returns the number of articles in the database.

XQuery:
```
let $a := input()/article
return count($a)
```

AmosQL:
```
count(
   select in(r0)
   from bag of node l0, bag of node r0
   where l0=(select evaluate(x, "/article ") from node x)
   and r0=l0
);
```

The statement for `f0` selects all `articles`. The statement for `r0` also select all articles since the root node of the evaluate call is `f0` and the path is ".". After building the query, the `count` function is applied on the results of the query (all articles).

3.2.5 Multiple `for` clauses

Using multiple `for` clauses are mostly used to have some kind of relationship between the different clauses. One possibility to express this relationship is directly in the `for` clause. The following example illustrates this.

TC/MD_Q4:
Find the heading of the section following the section entitled "Introduction" in a certain article with id attribute value (4).

XQuery:
```
for $a in
input()/article[@id="4"]/body/section[@heading="introduction"],
$p in input()/article[@id="4"]/body/section[. >> $a][1]
```

```
return
   <HeadingOfSection>
      {$p/@heading}
   </HeadingOfSection>
```

AmosQL:

```
select r0
from node f0, node xf0, node f1, node xf1, attr r0
where f0=evaluate(xf0,
   "/article[@id='4']/body/section[@heading='introduction']")
and f1=evaluate(f0, "following-sibling::node()[1] ")
and f1=evaluate(xf1, "/article[@id='4']/body/section")
and nodeName(r0)="heading"
and ownerElement(r0)=f1;
```

In this example the first section following a certain `section` with an attribute heading and value "introduction" should be returned. First, the `section` with the attribute `heading` and the value "introduction" is bound to `f0`. Then two expressions are used to get the relationship. The first expression of `f1` returns all sections in the article (with id value 4). The second one returns the first (`[1]`) section following (`following-sibling::node()`) the section returned by `f0`. The root node of this expression is `f0`.
Looking at the return statement, the special treatment of an attribute as return value is shown. Since attributes are not stored as nodes, `attr r0` has to be inserted into the type extents instead of `node r0`. An attribute always belongs to an element. Therefore the function `ownerElement(attr a) -> node n` is called. Since several attributes can belong to one element, the name of the attribute has to be checked using the `nodeName(attr a) -> charstring` function.

Another possibility of getting some kind of relationship between several `for` clauses is to use the `where` clause.

TC/MD_Q19:
Retrieve the headwords of entries cited, in etymology part, by certain entry with id attribute value (E1).

XQuery:

```
for $a in input()/article[@id="7"]/epilog/references/a_id,
   $b in input()/article
where $a = $b/@id
return
   <Output>
      {$b/prolog/title}
   </Output>
```

AmosQL:

```
select r0
from node f0, node xf0, node f1, node xf1, text tf0, node zf0,
   attr zf1, node r0
where f0=evaluate(xf0, "/article[@id='7']/epilog/references/a_id")
and f1=evaluate(xf1, "/article")
and zf0=evaluate(f0, "/.")
and parentNode(tf0)=zf0
```

```
and nodeName(zf1)="id"
and ownerElement(zf1)=f1
and nodeValue(tf0)=nodeValue(zf1)
and r0=evaluate(f1, "/prolog/title");
```

In this case temporary variables named `zf0` and `zf1` are introduced to express the condition of the `where` clause. Furthermore, a type extent `text` is introduced to match the string representation of the node value of `node`. The function `nodeValue` applied on a node does not return the string representation of this node. Therefore the `text` type extent has to be introduced. Applying the `nodeValue` function on an attribute returns the string representation and no `text` type extent is needed. `zf0` contains the nodes named `a_id`, which were already selected by `f0`. This expression is introduced because in case that the path does consist of several steps, they are performed here. `zf0` contains the parent nodes of the `text` type extent. `zf1` is of type `attr` and contains the `id` attribute of the nodes selected by `f0`.

3.2.6 Combination of `for` and `let` clause

This section will deal with the combination of a `for` and a `let` clause which is used by some queries in the waterloo benchmark.
There are basically two different kinds of combination of these two clauses. The first one introduced here is to get the count of a bag of objects.

TC/MD_Q3:
Group articles by date and calculate the total number of articles in each group.

XQuery:
```
for $a in distinct-values (input()/article/prolog/dateline/date)
let $b := input()/article/prolog/dateline[date=$a]
return
  <Output>
    <Date>{$a/text()}</Date>
    <NumberOfArticles>{count($b)}</NumberOfArticles>
  </Output>
```

AmosQL:
```
select distinct r0, count(r1)
from node f0, node xf0, bag of node l0, text f0t, node r0, bag of
  node r1
where f0=evaluate(xf0, "/article/prolog/dateline/date")
and l0=(select x from node x, text t, node x1 where x=evaluate(x1,
  "/article/prolog/dateline/date")
  and nodeValue(t)=nodeValue(f0t) and parentNode(t)=x)
and parentNode(f0t)=f0
and r0=evaluate(f0, "/text()")
and r1=l0;
```

The `for` clause selects all `disctinct` values of the nodes called "`date`". The `let` clause selects in each iteration of the `for` clause all the `datelines` who have the same `date` as the `date` selected by the `for` clause.
To express the different kind of variable bindings of the `let` clause compared to the `for` clause, an inner query is used. This inner query also uses a `text` type

extent to get the string representation of the `date`. The type extent of the corresponding expression of the let clause is `bag of node` and not simply `node` as in case of the `for` clause. Therefore the function `count(bag of object o) -> integer` can be applied and will return the number of objects in this bag.

The other possibility is very similar to the combination of two for clauses. The example given here will show it.

DC/MD_Q4
List the item id of the previous item of a matching item with id attribute value (8).

XQuery:
```
let $item := input()/items/item[@id="8"]
for $prevItem in input()/items/item[. << $item]
  [position() = last()]
return
  <Output>
    <CurrentItem>{$item/@id}</CurrentItem>
    <PreviousItem>{$prevItem/@id}</PreviousItem>
  </Output>
```

AmosQL:
```
select r0, r1
from node f0, node xf0, bag of node l0, attr r0, attr r1
where f0=evaluate(in(l0), "preceding-
  sibling::node()[position()=last()]")
and f0=evaluate(xf0, "/items/item")
and l0=(select evaluate(x, "/items/item[@id='8'] ") from node x)
and nodeName(r0)="id"
and ownerElement(r0)=in(l0)
and nodeName(r1)="id"
and ownerElement(r1)=f0;
```

As mentioned before there is no big difference between the combination of two `for` clauses and the combination of a `for` clause and a `let` clause as shown here.
The type extent of the let clause is still a bag of node (`l0`). Later, the function `in(bag of object o) -> bag of object x` is applied on `l0`. Therefore, at this point, there is no difference in treatment between `for` and `let` clauses. The only difference is that in case on of the element extracted by the `let` clause should be used as a root node of an evaluation, the function `in(bag of object o) -> bag of object x` has to be applied (first condition in the AmosQL `where` clause). If the `in(…)` function is not applied during evaluation like here through the `ownerElement(r0)=in(l0)` and the results of the `let` clause are directly results of the query, the `in(…)` function is applied "at the end" in the `select` statement (e.g. select in(r0)).

Since the direct translation of the variable binding done by the `let` clause is not possible, it is mostly treated the same as a `for` clause.
The difference is only obvious when using the `count(…)` function.

3.3 Restrictions

As mentioned before, not all expressions of the XQuery language can be translated into AmosQL statements. Nevertheless almost all queries of the Waterloo benchmark can be translated.

One that could not be translated was for example DC/SD_Q10.

```
for $a in input()/catalog/item
where $a/date_of_release gt "1990-01-01" and
   $a/date_of_release lt "1995-01-01"
order by $a/publisher/name
return
   <Output>
      {$a/title}
      {$a/publisher}
   </Output>
```

The results of this query should be ordered by the name of the publisher. In the current version of this project, only `order by` clauses which are also mentioned in the `return` clause can be evaluated. In the return clause, only publisher is returned, not his name. Therefore the translation is not possible.

Another one that could not be translated is DC/SD_Q20.

```
for $size in input()/catalog/item/attributes/size_of_book
where $size/length*$size/width*$size/height > 500000
return
   <Output>
      {$size/../../title}
   </Output>
```

In this query a calculation of an integer is done in the where clause (`$size/length*$size/width*$size/height > 500000`). This is not supported in the current version.

Another problem is the evaluation of XPath expressions with the AXE system. This system does not support character escapes (like 'eq' instead of '=') according to the XML rules. Therefore characters like comparison characters have to be written in their normal representation and not using their escape characters.

4. Translation Rules

A brief summary of the necessary steps to perform the translation is given here. For the translation of multiple for/let clauses with references to each other please see 3.2.2.

For every `for` statement do the following:

- insert a new `node` type extent into the from clause
- insert a `node x` type extent into the from clause for temporary results
- insert a new statement looking like
  `varName=evaluate(x, "path")` into the where clause

For every `let` statement do the following:

- insert a new `bag of node` type extent into the from clause
- insert a `node x` type extent into the from clause for temporary results
- insert a new statement looking like
  `varName=evaluate(x, "path")` into the where clause

For every `where` statement do the following:

- simple case (`where … = "…"`)
  insert into the where clause:
  `refVarName=evaluate(refVarName, "/.[predicate]")`
- empty(…)
  insert into the where clause:
  `notany(evaluate(refVar, "path"))`
- contains(…)
  insert inner query into the where clause (see 3.2.2)
- some …
  insert `some(inner query)` into the where clause (see 3.2.2)
- every …
  insert
  `count(innerQuery)=count(innerQueryNoConditions)`
  into the where clause (see 3.2.2)

For the `order by` statement do the following:

- write query as `sort(query, pos, 'inc'|'dec')`
  where `query` is the original query, `pos` is the position of the order key and `'inc'|'dec'` is the sort order

For every `return` statement do the following:

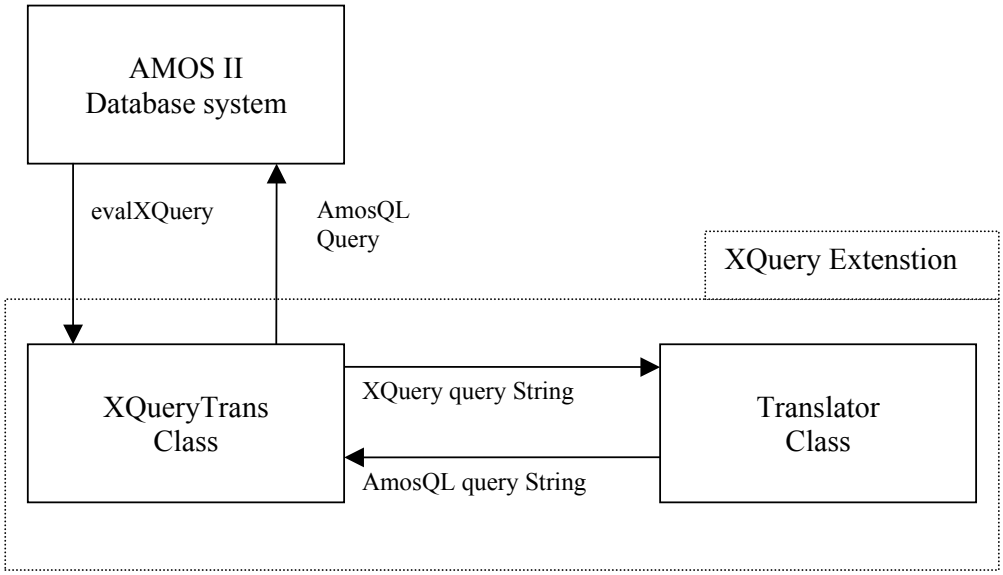
- insert a new `node/bag of node/attr` type extent into the from clause, depending on the referenced variable
- insert `varName/in(varName)/count(varName)` into the select clause
- if a path expression is to be returned:
  insert a new statement looking like
  `varName=evaluate(refVarName, "path")`

into the where clause, the `refNameVar` was declared in some `for` or `let` statement

- if an attribute is to be returned:
  insert
  `ownerElement(varName) = refVarName`
  and
  `nodeName(varName) = "nodeValue"`
  into the where clause
- if count(bag of object) is to be returned:
  insert
  `varName = evaluate(refVarName, "path")` into the where clause
  and
  if count is the only return value:
  rewrite query as `count(query)`
  if count is not the only return value:
  add `count(varName)` to the select clause

5. Implementation of the query translator

The query translater implemented in this project consist of two Java classes. The Translator class does the actual translation of the XQuery expression to the AmosQL expression. The XQueryTrans class is the driver class for the communication with the AMOS II database.



The function evalXQuery(String strXQueryExpression) of the XQueryTrans class takes the XQuery expression as argument and calls the Translator class. The Translator class does the actual translation to an AmosQL expression. This expression is returned to the XQueryTrans class which calls AMOS II and executes the AmosQL query.

To use this extension in AMOS II the following statement has to be executed:

```
create function evalxquery(Charstring expression) -> bag of Node
       as foreign "JAVA:XQueryTrans/evalXQuery";
```

With this statement a function called evalxquery is added to the AMOS II database system. It takes the XQuery expression as an argument. The return value is bag of node. The definition as foreign function is done by the statement `as foreign "JAVA:XQueryTrans/evalXQuery`. The keyword `JAVA` means that the extension is written in java. The name of the java class is `XQueryTrans` and the name of the function to call is `evalXQuery`.

After executing this statement the XQuery extension can be used as a regular function of AMOS II.

6. Conclusion and future work

XML is nowadays very common in use and it is therefore necessary to find a way to query the data of this format. Intelligent queries are necessary to deal with the large amount of data that needs to be processed.

In this paper, ways to translate XQuery expressions into queries of the Functional Database system AMOS II with its query language AmosQL were discussed. The first approach using an XMLSchema translator developed at UDBL was not able to deal with the whole scope of the XQuery queries. That system translated XML Schema definitions into corresponding AMOS II database schema definitions. Because of the data centric approach assumed by the XMLSchema translator, i.e. focussing on the storage of the data rather than on the structure of the document, constructs of XQuery dealing with the document centric elements could not be translated. In general XQuery expressions are over the document structure, not the XMLSchema data schema and therefore the translated XMLSchema is of no use for XQuery translation.

The other approach using the AXE system developed at the UDBL was capable to deal with both data-centric and document-centric XML documents and the corresponding queries. Due to the large complexity of XQuery, restrictions have to be made to build a translator. These restrictions where made by focusing only on the queries given by the Waterloo benchmark. Some more restrictions have to be made since they are not supported in AMOS II, for example the restrictions concerning the order by clause.

A complete query translator was implemented. This system is able to translate queries expressed in XQuery into corresponding queries in AmosQL using the database schema and functions given by the AXE system.

This implementation leaves some room for optimization and simplification of the generated expressions. This can be done as a future work.

Some restrictions had to be made to implement the translation. This can be a starting point for future work.
The restrictions that had to be made when translating the queries of the Waterloo benchmark concern the order by clause of a FLWOR expression and the calculation of values within a where clause.
The biggest restriction that had to be made concerns disallowing nested FLWOR expressions such as a FLWOR expression in a return clause. Such expressions are not present in the Waterloo benchmark.

Utilization of indexes is always wanted in a database environment. Since functions are needed to have indexes, it is not possible to use this mechanism in the current version. Future work can therefore use the first approach given in chapter 2 and combine it with the current version of this project to a hybrid system. The first approach was not implemented yet and can be done in the future.

Currently no syntax check is made for the queries expressed in XQuery. This could be added in future versions.

7. References

[1] Risch, T.; Josifovski, V.; Katchaounov, T. (2000):
AMOS II Concepts
http://www.dis.uu.se/~udbl/amos/doc/amos_concepts.html

[2] Risch, T.; Josifovski, V.; Katchaounov, T. (2000):
Functional Data Integration in a Distributed Mediator System
http://user.it.uu.se/~torer/publ/FuncMedPaper.pdf

[3] Elin, D.; Risch, T.(2000):
Amos II Java Interfaces
http://user.it.uu.se/~torer/publ/javaapi.pdf

[4] Bin Yao, B.; Ozsu, M.T.; Keenleyside, J. (2003)
XBench - A Family of Benchmarks for XML DBMSs
http://db.uwaterloo.ca/~ddbms/projects/xbench/

[5] Bray, T.; Paoli, J.; Sperberg-McQueen, C.M.; Maler, E. (2000):
Extensible Markup Language (XML) 1.0 (Second Edition)
W3C Recommendation
http://www.w3.org/TR/REC-xml

[6] Thompson, H.S.; Beech, D.; Maloney, M.; Mendelsohn, N. (2001):
XML Schema Part 1: Structures
W3C Recommendation
http://www.w3.org/TR/xmlschema-1/

[7] Clark, J.; DeRose, S. (1999)
XML Path Language (XPath) Version 1.0
W3C Recommendation
http://www.w3.org/TR/xpath

[8] Boag, S.; Chamberlin, D.; Fernández, M.F.; Florescu, D.; Robie, J.; Siméon, J. (2003):
XQuery 1.0: An XML Query Language
W3C Working Draft
http://www.w3.org/TR/xquery/

[9] Johannson, T.; Heggbrenna, R. (2003):
Importing XML Schema into an Object-Oriented Database Mediator System
Uppsala Master's Theses in Computer Science no. 260

Appendix A: Translated Queries

A.1 TC/MD

TC/MD_Q1:
Return the title of the article that has matching id attribute value (1)

```
for $art in input()/article[@id="1"]
return
  $art/prolog/title
```

AmosQL:

```
select r0
from node f0, node xf0, node r0
where f0=evaluate(xf0, "/article[@id='1']")
and r0=evaluate(f0, "/prolog/title");
```

TC/MD_Q2:
Find the title of the article authored by “Ben Yang”

```
for $prolog in input()/article/prolog
where $prolog/authors/author/name="Ben Yang"
return
  $prolog/title
```

AmosQL:

```
select r0
from node f0, node xf0, node tmp_f0, node r0
where f0=evaluate(xf0, "/article/prolog")
and tmp_f0=evaluate(f0, "/.[authors/author/name='Ben Yang']")
and r0=evaluate(tmp_f0, "/title");
```

TC/MD_Q3:
Group articles by date and calculate the total number of articles in each group

```
for $a in distinct-values (input()/article/prolog/dateline/date)
let $b := input()/article/prolog/dateline[date=$a]
return
  <Output>
    <Date>{$a/text()}</Date>
    <NumberOfArticles>{count($b)}</NumberOfArticles>
  </Output>
```

AmosQL:

```
select distinct r0, count(r1)
from node f0, node xf0, bag of node l0, text f0t, node r0, bag of
  node r1
where f0=evaluate(xf0, "/article/prolog/dateline/date")
and l0=(select x from node x, text t, node x1 where x=evaluate(x1,
  "/article/prolog/dateline/date")
  and nodeValue(t)=nodeValue(f0t) and parentNode(t)=x)
and parentNode(f0t)=f0
and r0=evaluate(f0, "/text()")
and r1=l0;
```

TC/MD_Q4:
Find the heading of the section following the section entitled “Introduction" in a certain article with id attribute value (4).

```
for $a in
input()/article[@id="4"]/body/section[@heading="introduction"],
$p in input()/article[@id="4"]/body/section[. >> $a][1]
return
  <HeadingOfSection>
    {$p/@heading}
  </HeadingOfSection>
```

AmosQL:

```
select r0
from node f0, node xf0, node f1, node xf1, attr r0
where f0=evaluate(xf0,
  "/article[@id='4']/body/section[@heading='introduction']")
and f1=evaluate(f0, "following-sibling::node()[1] ")
and f1=evaluate(xf1, "/article[@id='4']/body/section")
and nodeName(r0)="heading"
and ownerElement(r0)=f1;
```

TC/MD_Q5:

Return the headings of the first section of a certain article with id attribute value (2).

```
for $a in input()/article[@id="2"]
return
  <HeadingOfSection>
    {$a/body/section[1]/@heading}
  </HeadingOfSection>
```

AmosQL:

```
select r0
from node f0, node xf0, node r0
where f0=evaluate(xf0, "/article[@id='2']")
and r0=evaluate(f0, "/body/section[1]/@heading");
```

TC/MD_Q6:

Find titles of articles where both keywords (“the" and “hockey") are mentioned in the same paragraph of abstracts.

```
for $a in input()/article
where some $b in $a/body/abstract/p satisfies (contains($b, "the")
and contains($b, "hockey"))
return
  $a/prolog/title
```

AmosQL:

```
select r0
from node f0, node xf0, node w, node r0
where f0=evaluate(xf0, "/article")
and some(select x from node x, text t where x=evaluate(w, "/.")
  and like(nodeValue(t), "*the*") and parentNode(t)=x)
and some(select x from node x, text t where x=evaluate(w, "/.")
  and like(nodeValue(t), "*hockey*") and parentNode(t)=x)
and w=evaluate(f0, "/body/abstract/p")
and r0=evaluate(f0, "/prolog/title");
```

TC/MD_Q7:

Find titles of articles where a keyword (“hockey") is mentioned in every paragraph of abstract.

```
for $a in input()/article
where every $b in $a/body/abstract/p satisfies contains($b,
"hockey")
```

```
return
  $a/prolog/title
```

AmosQL:

```
select r0
from node f0, node xf0, node r0
where f0=evaluate(xf0, "/article")
and count(select x from node x, text t where x=evaluate(f0,
  "/body/abstract/p")and like(nodeValue(t), "*hockey*") and
  parentNode(t)=x)
=count(select x from node x where x=evaluate(f0,
  "/body/abstract/p"))
and r0=evaluate(f0, "/prolog/title");
```

TC/MD_Q8:

Return the names of all authors (one element name unknown) of the article with matching id attribute value (2).

```
for $art in input()/article[@id="2"]
return
  $art/prolog/*/author/name
```

AmosQL:

```
select r0
from node f0, node xf0, node r0
where f0=evaluate(xf0, "/article[@id='2']")
and r0=evaluate(f0, "/prolog/*/author/name");
```

TC/MD_Q9:

Return all author names (several consecutive element unknown) of the article with matching id attribute value (2).

```
for $art in input()/article[@id="2"]
return
  $art//author/name
```

AmosQL:

```
select r0
from node f0, node xf0, node r0
where f0=evaluate(xf0, "/article[@id='2']")
and r0=evaluate(f0, "//author/name");
```

TC/MD_Q10:

List the titles of articles sorted by country.

```
for $a in input()/article/prolog
order by $a/dateline/country
return
  <Output>
    {$a/title}
    {$a/dateline/country}
  </Output>
```

AmosQL:

```
sort((select r0, r1
  from node f0, node xf0, node r0, node r1
  where f0=evaluate(xf0, "/article/prolog")
  and r0=evaluate(f0, "/title")
  and r1=evaluate(f0, "/dateline/country")
), 2, 'inc');
```

TC/MD_Q11:
List the titles of articles that have a matching country element type (Canada), sorted by date.

```
for $a in input()/article/prolog
where $a/dateline/country="Canada"
order by $a/dateline/date
return
   <Output>
      {$a/title}
      {$a/dateline/date}
   </Output>
```

AmosQL:

```
sort((select r0, r1
   from node f0, node xf0, node tmp_f0, node r0, node r1
   where f0=evaluate(xf0, "/article/prolog")
   and tmp_f0=evaluate(f0, "/.[dateline/country='Canada']")
   and r0=evaluate(tmp_f0, "/title")
   and r1=evaluate(tmp_f0, "/dateline/date")
), 2, 'inc');
```

TC/MD_Q12:
Retrieve the body of the article that has a matching id attribute value (4).

```
for $a in input()/article[@id="4"]
return
   <Article>
      {$a/body}
   </Article>
```

AmosQL:

```
select r0
from node f0, node xf0, node r0
where f0=evaluate(xf0, "/article[@id='4']")
and r0=evaluate(f0, "/body");
```

TC/MD_Q13:
Construct a brief information on the article that has a matching id attribute value (5), including title, the name of _rst author, date and abstract.

```
for $a in input()/article[@id="5"]
return
   <Output>
      {$a/prolog/title}
      {$a/prolog/authors/author[1]/name}
      {$a/prolog/dateline/date}
      {$a/body/abstract}
   </Output>
```

AmosQL:

```
select r0, r1, r2, r3
from node f0, node xf0, node r0, node r1, node r2, node r3
where f0=evaluate(xf0, "/article[@id='5']")
and r0=evaluate(f0, "/prolog/title")
and r1=evaluate(f0, "/prolog/authors/author[1]/name")
and r2=evaluate(f0, "/prolog/dateline/date")
and r3=evaluate(f0, "/body/abstract");
```

TC/MD_Q14:
List article title that doesn't have genre element.

```
for $a in input()/article/prolog
```

```
where empty ($a/genre)
return
  <NoGenre>
    {$a/title}
  </NoGenre>
```

AmosQL:

```
select r0
from node f0, node xf0, node r0
where f0=evaluate(xf0, "/article/prolog")
and notany(evaluate(f0, "/genre"))
and r0=evaluate(f0, "/title");
```

TC/MD_Q15:

List author names whose contact elements are empty in articles.

```
for $a in input()/article/prolog/authors/author
where empty($a/contact/text())
return
  <NoContact>
    {$a/name}
  </NoContact>
```

AmosQL:

```
select r0
from node f0, node xf0, node r0
where f0=evaluate(xf0, "/article/prolog/authors/author")
and notany(evaluate(f0, "/contact/text()"))
and r0=evaluate(f0, "/name");
```

TC/MD_Q16:

Get the article by its id attribute value (4).

```
for $a in input()/article[@id="4"]
return
  $a
```

AmosQL:

```
select r0
from node f0, node xf0, node r0
where f0=evaluate(xf0, "/article[@id='4']")
and r0=evaluate(f0, "/.");
```

TC/MD_Q17:

Return the titles of articles which contain a certain word (“hockey").

```
for $a in input()/article
where contains ($a//p, "hockey")
return
  $a/prolog/title
```

AmosQL:

```
select r0
from node f0, node xf0, node r0
where f0=evaluate(xf0, "/article")
and some(select x from node x, text t where x=evaluate(f0, "//p")
and like(nodeValue(t), "*hockey*") and parentNode(t)=x)
and r0=evaluate(f0, "/prolog/title");
```

TC/MD_Q18:

List the titles and abstracts of articles which contain a given phrase (“the hockey").

```
for $a in input()/article
```

```
where contains ($a//p, "the hockey")
return
  <Output>
    {$a/prolog/title}
    {$a/body/abstract}
  </Output>
```

AmosQL:

```
select r0, r1
from node f0, node xf0, node r0, node r1
where f0=evaluate(xf0, "/article")
and some(select x from node x, text t where x=evaluate(f0, "//p")
  and like(nodeValue(t), "*the hockey*") and parentNode(t)=x)
and r0=evaluate(f0, "/prolog/title")
and r1=evaluate(f0, "/body/abstract");
```

TC/MD_Q19:

List the names of articles cited by an article with a certain id attribute value (2).

```
for $a in input()/article[@id='2']/epilog/references/a_id,
$b in input()/article
where $a = $b/@id
return
  <Output>
    {$b/prolog/title}
  </Output>
```

AmosQL:

```
select r0
from node f0, node xf0, node f1, node xf1, text tf0, node zf0,
  attr zf1, node r0
where f0=evaluate(xf0, "/article[@id='2']/epilog/references/a_id")
and f1=evaluate(xf1, "/article")
and zf0=evaluate(f0, "/.")
and parentNode(tf0)=zf0
and nodeName(zf1)="id"
and ownerElement(zf1)=f1
and nodeValue(tf0)=nodeValue(zf1)
and r0=evaluate(f1, "/prolog/title");
```

A.2 TC/SD

TC/SD_Q1:

Return the entry that has matching headword ("the").

```
for $ent in input()/dictionary/e
where $ent/hwg/hw="the"
return
  $ent
```

AmosQL:

```
select r0
from node f0, node xf0, node tmp_f0, node r0
where f0=evaluate(xf0, "/dictionary/e")
and tmp_f0=evaluate(f0, "/.[hwg/hw='the']")
and r0=evaluate(tmp_f0, "/.");
```

TC/SD_Q2:

Find the headword of the entry which has matching quotation year (1900).

```
for $ent in input()/dictionary/e
```

```
where $ent/ss/s/qp/q/qd="1900"
return
  $ent/hwg/hw
```

AmosQL:

```
select r0
from node f0, node xf0, node tmp_f0, node r0
where f0=evaluate(xf0, "/dictionary/e")
and tmp_f0=evaluate(f0, "/.[ss/s/qp/q/qd='1900']")
and r0=evaluate(tmp_f0, "/hwg/hw");
```

TC/SD_Q3:

Group entries by quotation location in a certain quotation year (1900) and calculate the total number entries in each group.

```
for $a in distinct-values
(input()/dictionary/e/ss/s/qp/q[qd="1900"]/loc)
let $b := input()/dictionary/e/ss/s/qp/q[loc=$a]
return
  <Output>
    <Location>{$a/text()}</Location>
    <NumberOfEntries>{count($b)}</NumberOfEntries>
  </Output>
```

AmosQL:

```
select distinct r0, count(r1)
from node f0, node xf0, bag of node l0, text f0t, node r0, bag of
  node r1
where f0=evaluate(xf0, "/dictionary/e/ss/s/qp/q[qd='1900']/loc")
and l0=(select x from node x, text t, node x1 where x=evaluate(x1,
  "/dictionary/e/ss/s/qp/q/loc") and nodeValue(t)=nodeValue(f0t)
  and parentNode(t)=x)
and parentNode(f0t)=f0
and r0=evaluate(f0, "/text()")
and r1=l0;
```

TC/SD_Q4:

List the headword of the previous entry of a matching headword ("you").

```
let $ent := input()/dictionary/e[hwg/hw="you"]
for $prevEnt in input()/dictionary/e[hwg/hw << $ent]
  [position() = last()]
return
  <Output>
    <CurrentEntry>{$ent/hwg/hw/text()}</CurrentEntry>
    <PreviousEntry>{$prevEnt/hwg/hw/text()}</PreviousEntry>
  </Output>
```

AmosQL:

```
select r0, r1
from node f0, node xf0, node zf0, bag of node l0, node r0, node r1
where zf0=evaluate(in(l0), "hwg/hw")
and zf0=evaluate(in(l0), "hwg/hw/preceding-
  sibling::node()[position() = last()]")
and f0=evaluate(xf0, "/dictionary/e")
and l0=(select evaluate(x, "/dictionary/e[hwg/hw='you'] ") from
  node x)
and r0=evaluate(in(l0), "/hwg/hw/text()")
and r1=evaluate(f0, "/hwg/hw/text()");
```

TC/SD_Q5:

Return the first sense of a matching headword ("that").

```
for $a in input()/dictionary/e
where $a/hwg/hw="that"
return
  $a/ss/s[1]
```

AmosQL:

```
select r0
from node f0, node xf0, node tmp_f0, node r0
where f0=evaluate(xf0, "/dictionary/e")
and tmp_f0=evaluate(f0, "/.[hwg/hw='that']")
and r0=evaluate(tmp_f0, "/ss/s[1]");
```

TC/SD_Q6:

Return the words where some quotations were quoted in a certain year (1900).

```
for $word in input()/dictionary/e
where some $item in $word/ss/s/qp/q
  satisfies $item/qd eq "1900"
return
  $word
```

AmosQL:

```
select r0
from node f0, node xf0, node w, node xw, node r0
where f0=evaluate(xf0, "/dictionary/e")
and xw=evaluate(w, "/.[qd eq '1900']")
and w=evaluate(f0, "/ss/s/qp/q")
and r0=evaluate(f0, "/.");
```

TC/SD_Q7:

Return the words where all quotations were quoted in a certain year (1900).

```
for $word in input()/dictionary/e
where every $item in $word/ss/s/qp/q
  satisfies $item/qd eq "1900"
return
  $word
```

AmosQL:

```
select r0
from node f0, node xf0, node r0
where f0=evaluate(xf0, "/dictionary/e")
and count(evaluate(f0, "/ss/s/qp/q[qd eq '1900']"))
=count(evaluate(f0, "/ss/s/qp/q"))
and r0=evaluate(f0, "/.");
```

TC/SD_Q8:

Return Quotation Text (one element name unknown) of a word ("and").

```
for $ent in input()/dictionary/e
where $ent/*/hw = "and"
return
  $ent/ss/s/qp/*/qt
```

AmosQL:

```
select r0
from node f0, node xf0, node tmp_f0, node r0
where f0=evaluate(xf0, "/dictionary/e")
and tmp_f0=evaluate(f0, "/.[*/hw = 'and']")
and r0=evaluate(tmp_f0, "/ss/s/qp/*/qt");
```

TC/SD_Q9:

Return Quotation Text (several consecutive element names unknown) of a word ("or").

```
for $ent in input()/dictionary/e
where $ent//hw = "or"
return
   $ent//qt
```

AmosQL:

```
select r0
from node f0, node xf0, node tmp_f0, node r0
where f0=evaluate(xf0, "/dictionary/e")
and tmp_f0=evaluate(f0, "/.[//hw = 'or']")
and r0=evaluate(tmp_f0, "//qt");
```

TC/SD_Q10:

List the words and their pronunciation, alphabetically, quoted in a certain year (1900).

```
for $a in input()/dictionary/e
where $a/ss/s/qp/q/qd = "1900"
order by $a/hwg/hw
return
   <Output>
      {$a/hwg/hw}
      {$a/hwg/pr}
   </Output>
```

AmosQL:

```
sort((select r0, r1
   from node f0, node xf0, node tmp_f0, node r0, node r1
   where f0=evaluate(xf0, "/dictionary/e")
   and tmp_f0=evaluate(f0, "/.[ss/s/qp/q/qd = '1900']")
   and r0=evaluate(tmp_f0, "/hwg/hw")
   and r1=evaluate(tmp_f0, "/hwg/pr")
), 1, 'inc');
```

TC/SD_Q11:

List the quotation locations and quotation dates, sorted by date, for a word ("word").

```
for $a in input()/dictionary/e[hwg/hw="the"]/ss/s/qp/q
order by $a/qd
return
   <Output>
      {$a/a}
      {$a/qd}
   </Output>
```

AmosQL:

```
sort((select r0, r1
   from node f0, node xf0, node r0, node r1
   where f0=evaluate(xf0, "/dictionary/e[hwg/hw='the']/ss/s/qp/q")
   and r0=evaluate(f0, "/a")
   and r1=evaluate(f0, "/qd")
), 2, 'inc');
```

TC/SD_Q12:

Retrieve the senses of a word ("his").

```
for $a in input()/dictionary/e
where $a/hwg/hw="his"
return
```

```
  <Entry>
    {$a/ss}
  </Entry>
```

AmosQL:

```
select r0
from node f0, node xf0, node tmp_f0, node r0
where f0=evaluate(xf0, "/dictionary/e")
and tmp_f0=evaluate(f0, "/.[hwg/hw='his']")
and r0=evaluate(tmp_f0, "/ss");
```

TC/SD_Q13:

Construct a brief information on a word ("his"), including: headword, pronunciation, part_of_speech, first etymology and first sense definition.

```
for $a in input()/dictionary/e
where $a/hwg/hw="his"
return
  <Output>
    {$a/hwg/hw}
    {$a/hwg/pr}
    {$a/hwg/pos}
    {$a/etymology/cr[1]}
    {$a/ss/s[1]/def}
  </Output>
```

AmosQL:

```
select r0, r1, r2, r3, r4
from node f0, node xf0, node tmp_f0, node r0, node r1, node r2,
  node r3, node r4
where f0=evaluate(xf0, "/dictionary/e")
and tmp_f0=evaluate(f0, "/.[hwg/hw='his']")
and r0=evaluate(tmp_f0, "/hwg/hw")
and r1=evaluate(tmp_f0, "/hwg/pr")
and r2=evaluate(tmp_f0, "/hwg/pos")
and r3=evaluate(tmp_f0, "/etymology/cr[1]")
and r4=evaluate(tmp_f0, "/ss/s[1]/def");
```

TC/SD_Q14:

List the ids of entries that do not have variant form lists and etymologies.

```
for $a in input()/dictionary/e
where empty($a/vfl) and empty($a/et)
return
  <NoVFLnET>
    {$a/@id}
  </NoVFLnET>
```

AmosQL:

```
select r0
from node f0, node xf0, attr r0
where f0=evaluate(xf0, "/dictionary/e")
and notany(evaluate(f0, "/vfl"))
and notany(evaluate(f0, "/et"))
and nodeName(r0)="id"
and ownerElement(r0)=f0;
```

TC/SD_Q17:

Return the headwords of the entries which contain a certain word ("hockey").

```
for $a in input()/dictionary/e
where contains ($a, "hockey")
return
  $a/hwg/hw
```

AmosQL:
```
select r0
from node f0, node xf0, node r0
where f0=evaluate(xf0, "/dictionary/e")
and some(select x from node x, text t where x=evaluate(f0, "/.")
   and like(nodeValue(t), "*hockey*") and parentNode(t)=x)
and r0=evaluate(f0, "/hwg/hw");
```

TC/SD_Q18:
List the headwords of entries which contain a given phrase ("the hockey").
```
for $a in input()/dictionary/e
where contains($a, "the hockey")
return
   $a/hwg/hw
```

AmosQL:
```
select r0
from node f0, node xf0, node r0
where f0=evaluate(xf0, "/dictionary/e")
and some(select x from node x, text t where x=evaluate(f0, "/.")
and like(nodeValue(t), "*the hockey*") and parentNode(t)=x)
and r0=evaluate(f0, "/hwg/hw");
```

TC/SD_Q19:
Retrieve the headwords of entries cited, in etymology part, by certain entry with id attribute value (E1).
```
for $ent in input()/dictionary/e[@id="E1"],
   $related in input()/dictionary/e
where $ent/et/cr = $related/@id
return
   <Output>
      {$related/hwg/hw}
   </Output>
```

AmosQL:
```
select r0
from node f0, node xf0, node f1, node xf1, text tf0, node zf0,
   attr zf1, node r0
where f0=evaluate(xf0, "/dictionary/e[@id='E1']")
and f1=evaluate(xf1, "/dictionary/e")
and zf0=evaluate(f0, "/et/cr")
and parentNode(tf0)=zf0
and nodeName(zf1)="id"
and ownerElement(zf1)=f1
and nodeValue(tf0)=nodeValue(zf1)
and r0=evaluate(f1, "/hwg/hw");
```

A.3 DC/MD

DC/MD_Q1:
Return the customer id of the order that has matching id attribute value (1).
```
for $order in input()/order[@id="1"]
return
$order/customer_id
```

AmosQL:

```
select r0
from node f0, node xf0, node r0
where f0=evaluate(xf0, "/order[@id='1']")
and r0=evaluate(f0, "/customer_id");
```

DC/MD_Q3:

Group orders with total amount bigger than a certain number(11000.0), by customer id and calculate the total number of each group.

```
for $a in distinct-values (input()/order
  [total > 11000.0]/customer_id)
let $b := input()/order[customer_id=$a]
return
  <Output>
    <CustKey>{$a/text()}</CustKey>
    <NumberOfOrders>{count($b)}</NumberOfOrders>
  </Output>
```

AmosQL:

```
select distinct r0, count(r1)
from node f0, node xf0, bag of node l0, text f0t, node r0, bag of
  node r1
where f0=evaluate(xf0, "/order[total > 11000.0]/customer_id")
and l0=(select x from node x, text t, node x1 where x=evaluate(x1,
  "/order/customer_id") and nodeValue(t)=nodeValue(f0t) and
  parentNode(t)=x)
and parentNode(f0t)=f0
and r0=evaluate(f0, "/text()")
and r1=l0;
```

DC/MD_Q4

List the item id of the previous item of a matching item with id attribute value (8).

```
let $item := input()/items/item[@id="8"]
for $prevItem in input()/items/item[. << $item][position()=last()]
return
  <Output>
    <CurrentItem>{$item/@id}</CurrentItem>
    <PreviousItem>{$prevItem/@id}</PreviousItem>
  </Output>
```

AmosQL:

```
select r0, r1
from node f0, node xf0, bag of node l0, attr r0, attr r1
where f0=evaluate(in(l0), "preceding-
  sibling::node()[position()=last()]")
and f0=evaluate(xf0, "/items/item")
and l0=(select evaluate(x, "/items/item[@id='8'] ") from node x)
and nodeName(r0)="id"
and ownerElement(r0)=in(l0)
and nodeName(r1)="id"
and ownerElement(r1)=f0;
```

DC/MD_Q5:

Return the first order line item of a certain order with id attribute value (2).

```
for $a in input()/order[@id="2"]
return
  $a/order_lines/order_line[1]
```

AmosQL:

```
select r0
```

```
from node f0, node xf0, node r0
where f0=evaluate(xf0, "/order[@id='2']")
and r0=evaluate(f0, "/order_lines/order_line[1]");
```

DC/MD_Q6:
Return invoice where some discount rates of sub-line items are higher than a certain number (0.02).

```
for $ord in input()/order
where some $item in $ord/order_lines/order_line
  satisfies $item/discount_rate gt 0.02
return
  $ord
```

AmosQL:

```
select r0
from node f0, node xf0, node w, node xw, node r0
where f0=evaluate(xf0, "/order")
and xw=evaluate(w, "/.[discount_rate gt 0.02]")
and w=evaluate(f0, "/order_lines/order_line")
and r0=evaluate(f0, "/.");
```

DC/MD_Q7:
Return invoice where all discount rates of sub-line items are higher than a certain number (0.02).

```
for $ord in input()/order
where every $item in $ord/order_lines/order_line
  satisfies $item/discount_rate gt 0.02
return
  $ord
```

AmosQL:

```
select r0
from node f0, node xf0, node r0
where f0=evaluate(xf0, "/order")
and count(evaluate(f0, "/order_lines/order_line[discount_rate gt
  0.02]"))
=count(evaluate(f0, "/order_lines/order_line"))
and r0=evaluate(f0, "/.");
```

DC/MD_Q8:
Return the order line item ids of an order with an attribute value (3).

```
for $a in input()/order[@id="3"]
return
  $a/*/order_line/item_id
```

AmosQL:

```
select r0
from node f0, node xf0, node r0
where f0=evaluate(xf0, "/order[@id='3']")
and r0=evaluate(f0, "/*/order_line/item_id");
```

DC/MD_Q9:
Return the item ids of an order with id attribute value (4).

```
for $a in input()/order[@id="4"]
return
  $a//item_id
```

AmosQL:

```
select r0
from node f0, node xf0, node r0
where f0=evaluate(xf0, "/order[@id='4']")
and r0=evaluate(f0, "//item_id");
```

DC/MD_Q10:
List the orders (order id, order date and ship type), with total amount larger than a certain number (11000.0), ordered alphabetically by ship type.

```
for $a in input()/order
where $a/total gt 11000.0
order by $a/ship_type
return
  <Output>
    {$a/@id}
    {$a/order_date}
    {$a/ship_type}
  </Output>
```

AmosQL:

```
sort((select r0, r1, r2
  from node f0, node xf0, node tmp_f0, attr r0, node r1, node r2
  where f0=evaluate(xf0, "/order")
  and tmp_f0=evaluate(f0, "/.[total gt 11000.0]")
  and nodeName(r0)="id"
  and ownerElement(r0)=tmp_f0
  and r1=evaluate(tmp_f0, "/order_date")
  and r2=evaluate(tmp_f0, "/ship_type")
), 3, 'inc');
```

DC/MD_Q11:
List the orders (order id, order date and order total), with total amount larger than a certain number (11000.0), in descending order by total amount.

```
for $a in input()/order
where $a/total gt 11000.0
order by $a/total descending
return
  <Output>
    {$a/@id}
    {$a/order_date}
    {$a/total}
  </Output>
```

AmosQL:

```
sort((select r0, r1, r2
  from node f0, node xf0, node tmp_f0, attr r0, node r1, node r2
  where f0=evaluate(xf0, "/order")
  and tmp_f0=evaluate(f0, "/.[total gt 11000.0]")
  and nodeName(r0)="id"
  and ownerElement(r0)=tmp_f0
  and r1=evaluate(tmp_f0, "/order_date")
  and r2=evaluate(tmp_f0, "/total")
), 3, 'dec');
```

DC/MD_Q12:
List all order lines of a certain order with id attribute value (5).

```
for $a in input()/order[@id="5"]
return
  <Output>
    {$a/order_lines}
```

```
  </Output>
```

AmosQL:

```
select r0
from node f0, node xf0, node r0
where f0=evaluate(xf0, "/order[@id='5']")
and r0=evaluate(f0, "/order_lines");
```

DC/MD_Q14:

List the ids of orders that only have one order line.

```
for $a in input()/order
where empty($a/order_lines/order_line[2])
return
  <OneItemLine>
    {$a/@id}
  </OneItemLine>
```

AmosQL:

```
select r0
from node f0, node xf0, attr r0
where f0=evaluate(xf0, "/order")
and notany(evaluate(f0, "/order_lines/order_line[2]"))
and nodeName(r0)="id"
and ownerElement(r0)=f0;
```

DC/MD_Q16:

Retrieve one whole order document with certain id attribute value (6).

```
for $a in input()/order[@id="6"]
return
  $a
```

AmosQL:

```
select r0
from node f0, node xf0, node r0
where f0=evaluate(xf0, "/order[@id='6']")
and r0=evaluate(f0, "/.");
```

DC/MD_Q17:

Return the ids of authors whose biographies contain a certain word ("hockey").

```
for $a in input()/authors/author
where contains ($a/biography, "hockey")
return
  <Output>
    {$a/@id}
  </Output>
```

AmosQL:

```
select r0
from node f0, node xf0, attr r0
where f0=evaluate(xf0, "/authors/author")
and some(select x from node x, text t where x=evaluate(f0,
  "/biography") and like(nodeValue(t), "*hockey*") and
  parentNode(t)=x)
and nodeName(r0)="id"
and ownerElement(r0)=f0;
```

DC/MD_19:

For a particular order with id attribute value (7), get its customer name and phone, and its order status.

```
for $order in input()/order, $cust in input()/customers/customer
where $order/customer_id = $cust/@id
  and $order/@id = "7"
return
  <Output>
    {$order/@id}
    {$order/order_status}
    {$cust/first_name}
    {$cust/last_name}
    {$cust/phone_number}
  </Output>
```

AmosQL:

```
select r0, r1, r2, r3, r4
from node f0, node xf0, node f1, node xf1, text tf0, node zf0,
  attr zf1, node tmp_f0, attr r0, node r1, node r2, node r3,
  node r4
where f0=evaluate(xf0, "/order")
and f1=evaluate(xf1, "/customers/customer")
and zf0=evaluate(f0, "/customer_id")
and parentNode(tf0)=zf0
and nodeName(zf1)="id"
and ownerElement(zf1)=f1
and nodeValue(tf0)=nodeValue(zf1)
and tmp_f0=evaluate(f0, "/.[@id = '7']")
and nodeName(r0)="id"
and ownerElement(r0)=tmp_f0
and r1=evaluate(tmp_f0, "/order_status")
and r2=evaluate(f1, "/first_name")
and r3=evaluate(f1, "/last_name")
and r4=evaluate(f1, "/phone_number");
```

A.4 DC/SD

DC/SD_Q1

Return the item that has matching item id attribute value (I1).

```
for $item in input()/catalog/item[@id="I1"]
return
  $item
```

AmosQL:

```
select r0
from node f0, node xf0, node r0
where f0=evaluate(xf0, "/catalog/item[@id='I1']")
and r0=evaluate(f0, "/.");
```

DC/SD_Q2

Find the title of the item which has matching author first name (Ben).

```
for $item in input()/catalog/item
where $item/authors/author/name/first_name = "Ben"
return
  $item/title
```

AmosQL:

```
select r0
from node f0, node xf0, node tmp_f0, node r0
where f0=evaluate(xf0, "/catalog/item")
```

```
and tmp_f0=evaluate(f0, "/.[authors/author/name/first_name =
   'Ben']")
and r0=evaluate(tmp_f0, "/title");
```

DC/SD_Q3:
Group items released in a certain year (1990), by publisher name and calculate the total number of items for each group.

```
for $a in distinct-values (input()/catalog/item
   [date_of_release >= "1990-01-01"]
   [date_of_release < "1991-01-01"]/publisher/name)
let $b := input()/catalog/item/publisher[name=$a]
return
   <Output>
      <Publisher>{$a/text()}</Publisher>
      <NumberOfItems>{count($b)}</NumberOfItems>
   </Output>
```

AmosQL:

```
select distinct r0, count(r1)
from node f0, node xf0, bag of node l0, text f0t, node r0, bag of
   node r1
where f0=evaluate(xf0, "/catalog/item
   [date_of_release >= '1990-01-01']
   [date_of_release < '1991-01-01']/publisher/name")
and l0=(select x from node x, text t, node x1 where x=evaluate(x1,
   "/catalog/item/publisher/name") and nodeValue(t)=nodeValue(f0t)
   and parentNode(t)=x)
and parentNode(f0t)=f0
and r0=evaluate(f0, "/text()")
and r1=l0;
```

DC/SD_Q4:
List the item id of the previous item of a matching item with id attribute value (I2).

```
let $item := input()/catalog/item[@id="I2"]
for $prevItem in input()/catalog/item
   [. << $item][position() = last()]
return
   <Output>
      <CurrentItem>{$item/@id}</CurrentItem>
      <PreviousItem>{$prevItem/@id}</PreviousItem>
   </Output>
```

AmosQL:

```
select r0, r1
from node f0, node xf0, bag of node l0, attr r0, attr r1
where f0=evaluate(in(l0), "preceding-sibling::node()[position() =
   last()]")
and f0=evaluate(xf0, "/catalog/item")
and l0=(select evaluate(x, "/catalog/item[@id='I2'] ") from node
   x)
and nodeName(r0)="id"
and ownerElement(r0)=in(l0)
and nodeName(r1)="id"
and ownerElement(r1)=f0;
```

DC/SD_Q5:
Return the information about the _rst author of item with a matching id attribute value (I3).

```
for $a in input()/catalog/item[@id="I3"]
return
  $a/authors/author[1]
```

AmosQL:

```
select r0
from node f0, node xf0, node r0
where f0=evaluate(xf0, "/catalog/item[@id='I3']")
and r0=evaluate(f0, "/authors/author[1]");
```

DC/SD_Q6:

Return item information where some authors are from certain country (Canada).

```
for $item in input()/catalog/item
where some $auth in
  $item/authors/author/contact_information/mailing_address
  satisfies $auth/name_of_country = "Canada"
return
  $item
```

AmosQL:

```
select r0
from node f0, node xf0, node w, node xw, node r0
where f0=evaluate(xf0, "/catalog/item")
and xw=evaluate(w, "/.[name_of_country = 'Canada']")
and w=evaluate(f0,
  "/authors/author/contact_information/mailing_address")
and r0=evaluate(f0, "/.");
```

DC/SD_Q7:

Return item information where all its authors are from certain country (Canada).

```
for $item in input()/catalog/item
where every $add in
  $item/authors/author/contact_information/mailing_address
  satisfies $add/name_of_country = "Canada"
return
  $item
```

AmosQL:

```
select r0
from node f0, node xf0, node r0
where f0=evaluate(xf0, "/catalog/item")
and count(evaluate(f0,
"/authors/author/contact_information/mailing_address[name
  _of_country = 'Canada']"))
=count(evaluate(f0,
  "/authors/author/contact_information/mailing_address"))
and r0=evaluate(f0, "/.");
```

DC/SD_Q8:

Return the publisher of an item with id attribute value (I4).

```
for $a in input()/catalog/*[@id="I4"]
return
  $a/publisher
```

AmosQL:

```
select r0
from node f0, node xf0, node r0
where f0=evaluate(xf0, "/catalog/*[@id='I4']")
and r0=evaluate(f0, "/publisher");
```

DC/SD_Q9:
Return the ISBN of an item with id attribute value (I5).

```
for $a in input()/catalog/item
where $a/@id="I5"
return
   $a//ISBN/text()
```

AmosQL:

```
select r0
from node f0, node xf0, node tmp_f0, node r0
where f0=evaluate(xf0, "/catalog/item")
and tmp_f0=evaluate(f0, "/.[@id='I5']")
and r0=evaluate(tmp_f0, "//ISBN/text()");
```

DC/SD_Q10:
List the item titles ordered alphabetically by publisher name, with
release date within a certain time period (from 1990-01-01 to 1995-01-01).

```
for $a in input()/catalog/item
where $a/date_of_release gt "1990-01-01" and
   $a/date_of_release lt "1995-01-01"
order by $a/publisher/name
return
   <Output>
      {$a/title}
      {$a/publisher}
   </Output>
```

AmosQL:

```
not supported in current version
unordered query generated:
select r0, r1
from node f0, node xf0, node tmp_f0, node tmp_tmp_f0, node r0,
   node r1
where f0=evaluate(xf0, "/catalog/item")
and tmp_f0=evaluate(f0, "/.[date_of_release gt '1990-01-01']")
and tmp_tmp_f0=evaluate(tmp_f0, "/.[date_of_release lt '1995-01-
   01']")
and r0=evaluate(tmp_tmp_f0, "/title")
and r1=evaluate(tmp_tmp_f0, "/publisher");
```

DC/SD_Q11:
List the item titles in descending order by date of release with
date of release within a certain time range (from 1990-01-01 to 1995-01-01).

```
for $a in input()/catalog/item
where $a/date_of_release gt "1990-01-01" and
   $a/date_of_release lt "1995-01-01"
order by $a/date_of_release descending
return
   <Output>
      {$a/title}
      {$a/date_of_release}
   </Output>
```

AmosQL:

```
sort((select r0, r1
   from node f0, node xf0, node tmp_f0, node tmp_tmp_f0, node r0,
      node r1
   where f0=evaluate(xf0, "/catalog/item")
```

```
   and tmp_f0=evaluate(f0, "/.[date_of_release gt '1990-01-01']")
   and tmp_tmp_f0=evaluate(tmp_f0, "/.
      [date_of_release lt '1995-01-01']")
   and r0=evaluate(tmp_tmp_f0, "/title")
   and r1=evaluate(tmp_tmp_f0, "/date_of_release")
), 2, 'dec');
```

DC/SD_Q12:
Get the mailing address of the _rst author of certain item with id attribute value (I6).

```
for $a in input()/catalog/item[@id="I6"]
return
   <Output>
      {$a/authors/author[1]/contact_information/mailing_address}
   </Output>
```

AmosQL:

```
select r0
from node f0, node xf0, node r0
where f0=evaluate(xf0, "/catalog/item[@id='I6']")
and r0=evaluate(f0,
   "/authors/author[1]/contact_information/mailing_address");
```

DC/SD_Q14:
Return the names of publishers who publish books between a period of time (from 1990-01-01 to 1991-01-01) but do not have FAX number.

```
for $a in input()/catalog/item
where $a/date_of_release gt "1990-01-01" and
   $a/date_of_release lt "1991-01-01" and
   empty($a/publisher/contact_information/FAX_number)
return
<Output>
   {$a/publisher/name}
</Output>
```

AmosQL:

```
select r0
from node f0, node xf0, node tmp_f0, node tmp_tmp_f0, node r0
where f0=evaluate(xf0, "/catalog/item")
and tmp_f0=evaluate(f0, "/.[date_of_release gt '1990-01-01']")
and tmp_tmp_f0=evaluate(tmp_f0, "/.[date_of_release lt '1991-01-
   01']")
and notany(evaluate(tmp_tmp_f0,
   "/publisher/contact_information/FAX_number"))
and r0=evaluate(tmp_tmp_f0, "/publisher/name");
```

DC/SD_Q17:
Return the ids of items whose descriptions contain a certain word ("hockey").

```
for $a in input()/catalog/item
where contains ($a/description, "hockey")
return
   <Output>
      {$a/@id}
   </Output>
```

AmosQL:

```
select r0
from node f0, node xf0, attr r0
where f0=evaluate(xf0, "/catalog/item")
```

```
and some(select x from node x, text t where x=evaluate(f0,
  "/description") and like(nodeValue(t), "*hockey*") and
  parentNode(t)=x)
and nodeName(r0)="id"
and ownerElement(r0)=f0;
```

DC/SD_Q19:

Retrieve the item titles related by certain item with id attribute value (I7).

```
for $item in input()/catalog/item[@id="I7"],
  $related in input()/catalog/item
where $item/related_items/related_item/item_id = $related/@id
return
  <Output>
  {$related/title}
  </Output>
```

AmosQL:

```
select r0
from node f0, node xf0, node f1, node xf1, text tf0, node zf0,
  attr zf1, node r0
where f0=evaluate(xf0, "/catalog/item[@id='I7']")
and f1=evaluate(xf1, "/catalog/item")
and zf0=evaluate(f0, "/related_items/related_item/item_id")
and parentNode(tf0)=zf0
and nodeName(zf1)="id"
and ownerElement(zf1)=f1
and nodeValue(tf0)=nodeValue(zf1)
and r0=evaluate(f1, "/title");
```

DC/SD_Q20:

Retrieve the item title whose size (length*width*height) is bigger than certain number (500000).

```
for $size in input()/catalog/item/attributes/size_of_book
where $size/length*$size/width*$size/height > 500000
return
  <Output>
    {$size/../../title}
  </Output>
```

AmosQL:

```
not supported in current version
query generated:
select r0
from node f0, node xf0, node tmp_f0, node r0
where f0=evaluate(xf0, "/catalog/item/attributes/size_of_book")
and tmp_f0=evaluate(f0, "/.[length*$size/width*$size/height >
  500000]")
and r0=evaluate(tmp_f0, "/../../title");
```