Extension Mechanism in Extensible XML Query Language X^2QL

Norihide Shinagawa
Center for Environmental Remote Sensing, Chiba University
1-33 Yayoi-cho, Inage-ku,
Chiba 263-8522, Japan
siena@cr.chiba-u.ac.jp

Hiroyuki Kitagawa
Institute of Information Sciences and Electronics, University of Tsukuba
1-1-1 Tennohdai, Tsukuba,
Ibaraki 305-8573, Japan
kitagawa@is.tsukuba.ac.jp

Abstract

XML has been used to code various types of data in a wide range of application domains, and the volume of XML data has been rapidly increasing. XML query languages provide a clue to manipulating huge amounts of XML data. XML data in each application domain have their own structures and associated semantics. Therefore, to appropriately process them, we sometimes need domain dependent functions, taking into account content characteristics. To utilize such functions in XML queries, we need extensibility because they can not be covered by built-in functions.

We have developed eXtensible XML Query Language X^2QL and its processing system, which feature user-defined foreign functions in the context of XML-QL-based query facilities. Foreign functions are implemented using general programming languages. Extensibility is based on an XML data model in which elements are modeled as stateful objects. This gives "dynamic aspects" (concepts of methods and processing-time properties) to elements, which have been treated as static data in general XML query languages. Our extensibility also enables user-defined aggregation functions and facilities such as sort and top-N selection.

This paper explains X^2QL and its extensibility. Focus is on the data model, Java binding of foreign functions, and how to implement them. The paper also describes development of an X^2QL query processor working with XML servers.

1. Introduction

With the rapid advance of the Internet, management of structured documents such as XML documents and their databases has become more and more important [1]-[3]. XML has been used to code various types of data in a wide range of application domains, and the volume of XML data has rapidly increased. XML query languages provide a clue to manipulating huge amounts of XML data [3].

XML-QL [4], XPath [5] and XQL [6] are well-known XML query languages, and XQuery [7] is an emerging XML query language developed by W3C. Although YATL [8], StruQL [9], Lorel [10] and UnQL [11] are proposed as query languages for semistructured data, XML documents are their potential targets. They allow the user to code queries in a declarative manner. XML-QL, XQuery and many query languages for semistructured data enable powerful tag-based data structure manipulation, such as the restructuring of element hierarchies and joins. However, they are very weak in content processing.

XML instances are organized for each application domain, and their contents have their own structures and associated semantics. Therefore, to appropriately process them, we sometimes need domain-dependent functions taking into account content characteristics. Examples include summary generation functions and similarity calculation functions for news article elements and feature extraction functions for elements associated with images. Various requirements and many techniques are needed to achieve each of them. Because of this variety, it is very difficult to provide a complete set of functions in advance. To use such functions in XML queries, we need extensibility. However, general XML query languages have little or no extensibility.

We developed eXtensible XML Query Language X^2QL [12] [13], which has extensibility via user-defined foreign functions and powerful tag-based data structure manipulation based on XML-QL [4]. Foreign functions are given as external programs written in general programming languages. By including appropriate user-defined foreign functions, each user can extend the processing power of X^2QL. This extensibility makes it possible to integrate application-oriented high-level content processing facilities into querying XML instances. Extensibility is based on an XML data model in which elements are modeled as stateful objects.

This gives "dynamic aspects" (methods and processing-time properties) to elements, which are treated as static data in general XML query languages. Our extensibility also enables user-defined aggregation functions, and facilities such as sort and top-N selection. This paper explains X^{2}QL and its extensibility. Focus is on the data model and Java binding to implement foreign functions. The paper also describes development of an X^{2}QL query processor working with XML servers.

Section 2 shows some query examples in which the inclusion of domain dependent contents processing is beneficial. Section 3 explains the main constructs of X^{2}QL and includes query examples. Section 4 describes the data model and the extension mechanism for foreign functions. Section 5 explains additional features of X^{2}QL. Section 6 describes the X^{2}QL processor. Section 7 mentions related work and Section 8 summarizes the paper.

2. Motivating Examples

This section shows two examples of XML queries in which the inclusion of domain dependent functions is beneficial. These query examples are specified in X^{2}QL in Section 3.

Suppose that an XML instance including newspaper articles is given in which each article consists of its publication date, category, headline, and news body. Its DTD is coded as follows:

```
<!-- DTD of the input document -->
<!ELEMENT Document  Article+>
<!ELEMENT Article  (Date,Category,
                    Headline,Body)>
<!ELEMENT Date     (Year,Month,Day)>
<!ELEMENT Year      #PCDATA>
<!ELEMENT Month     #PCDATA>
<!ELEMENT Day       #PCDATA>
<!ELEMENT Category  #PCDATA>
<!ELEMENT Headline  #PCDATA>
<!ELEMENT Body      Paragraph+>
<!ELEMENT Paragraph #PCDATA>.
```

Example 1 We generate a new XML instance in which articles published after 2000 are selected and grouped by category, and their summaries are added. The DTD of the output document is listed below. This query requires data structure manipulation to select `Article` elements published after 2000 and to group them by the `Category` values. It also requires the content processing to generate summaries of `Article` elements.

```
<!-- DTD of the output document -->
<!ELEMENT Document   DateGroup+>
<!ELEMENT Group      Article+>
<!ATTLIST Group      Category CDATA
                     #REQUIRED>
<!ELEMENT Article   (Date,Headline,
                     Summary,Body)>
<!ELEMENT Date      (Year,Month,Day)>
<!ELEMENT Year       #PCDATA>
<!ELEMENT Month      #PCDATA>
<!ELEMENT Day        #PCDATA>
<!ELEMENT Headline   #PCDATA>
<!ELEMENT Summary    #PCDATA>
<!ELEMENT Body       Paragraph+>
<!ELEMENT Paragraph  #PCDATA>
```

Example 2 We give a set of keywords and get the top N articles related to the keywords. The ranking order of each `Article` element is determined by similarity with the keywords. The similarity value is given by the content processing function.

3. X^{2}QL: An eXtensible XML Query Language

X^{2}QL features the inclusion of user-defined *foreign functions* [12] [13]. The implementation of foreign functions is accomplished through external programs. First, Subsection 3.1 gives the syntax of X^{2}QL. Then, basic foreign functions are explained in Subsection 3.2. Subsection 3.3 shows how example queries in Section 2 are specified in X^{2}QL.

3.1. Query Syntax

The syntax of X^{2}QL is based on XML-QL as follows:

```
where        patterns in source
         [, patterns in source]*
         [, predicate]*
[rank-by     ranking keys   top number]
[filter-by   filtering function calls]
[group-by    grouping keys]
[order-by    ordering keys [descending]]
construct    construction of each output
```

The `where`, `order-by`, and `construct` clauses are the same as in XML-QL, except that they can contain foreign function calls as explained in Subsection 3.2. The `group-by` clause is included based on discussion on future issues on XML-QL [4].

The `where` clause binds variables according to specified conditions. Variables are bound by `element_as` and `content_as` expressions, and embedded variables in *patterns* are bound to the corresponding values. For convenience, we call each set of variable bindings a *tuple*. Note that the outermost `where` clause can be omitted.

The `construct` clause generates the output for each tuple. When `ID` is specified in a tag, output elements having the same `ID` values in an element are merged into one element. `ID` values are usually given by Skolem functions. Note that `construct` clauses can have nested subqueries.

The semantics and processing order of optional clauses are as follows. The `rank-by` clause is used to select the top *number* tuples based on the ranking given by the *ranking keys*. This selection is performed immediately after the `where` clause. The order of selected tuples is not affected by this clause; the order of output elements is based on the order in the input. In text processing, it is often necessary to rank target elements by some similarity or importance measure derived by the foreign function, and to select the top N elements based on their ranks. To ease such processing, we have included the `rank-by` clause as an extension.

The `filter-by` clause is used to manipulate the sequence of tuples from the `rank-by` clause, and to generate a new one. This clause enables us to select and sort tuples based on characteristics derived from the whole sequence. Selection of the first $x\%$ tuples, based on the size of the sequence, is an example. This manipulation is performed by foreign functions called *filtering functions*. More details of filtering functions are given in Subsection 5.2. Note that the `rank-by` clause can be regarded as a specialized version of this clause introduced for convenience.

The `group-by` clause is used to group tuples passed from the `filter-by` clause based on *grouping keys* for aggregating calculations by foreign functions called *aggregation functions*. Aggregation functions are applied to each group. This clause does not change the order of tuples as well as the `order-by` clause. In an implementation, for each aggregation function call, a table of the grouping key values and aggregation values may be created by this clause and referred in output construction. More details of aggregation functions are given in Subsection 5.1.

The `order-by` clause is used to sort the output from the `construct` clause based on *ordering keys*. Note that this clause can be emulated by the `filter-by` clause with a filtering function call to sort tuples.

3.2. Foreign Functions

Functions defined in XML-QL are merely parameterized queries, so extensibility is insufficient. In X^2QL, users can define foreign functions whose implementations are given as external programs. In the current version of X^2QL, we assume that external programs are written in Java. See Sections 4 for details on foreign functions.

Basic foreign functions are classified into *general functions* and *element methods*. The other types of foreign functions are described in Section 5. General functions are normal-style functions defined in the global namespace. Elements, on the other hand, are conceptually regarded as objects, and each element method is associated with a specific element type, which means each is defined in its local namespace. Therefore, element methods of the same name can be defined for different element types. When a foreign function is called, the element type of the target element is checked, then an appropriate element method is invoked depending on the target element.

Let us consider foreign functions required for the examples in Section 2. For Example 1, we need an element method (named `abstract()`) associated with the `Article` element type to summarize each `Article` element and to return a `Summary` element. The element method generates a summary of an `Article` element taking into account the target dependent characteristics. This is done under the assumption that terms appearing in its `Headline` element are important.

For Example 2, we need a function to measure similarity between the given keywords and an element content. Here, for explanation purposes, we define it as a general function, `sim_cosine()`, which calculates similarities based on the traditional cosine measure [14]. By applying the function to an `Article` element, its similarity value is returned. [1]

In our environment, foreign functions are defined as follows:

```
function   type-name function-name( arg-list )
defined-by "URI of the implementation"


function   type-name element-type-name
                     .method-name( arg-list )
defined-by "URI of the implementation"


arg-list ::= type-name arg-name
         [, type-name arg-name]*.
```

In defining a foreign function, we specify the data types of arguments and return values. The data types `string`, `number`, `boolean`, `content`, `element` and all element types are allowed. Variables bound by `content_as` are associated with `content` and `element_as` are associated with `element` values.

The definitions of `abstract()` and `sim_cosine()` are coded as follows:

[1] Similarity measures often need document-dependent global factors (e.g. idf). The implementations of foreign functions are responsible for calculating them in their first invocation.

```
function  Summary Article.abstract()
defined-by "http://fqdn/path/
        pkg.ArticleMethods#abstract"

function  number  sim_cosine(
          content target,
          string  keyword )
defined-by "http://fqdn/path/
             common.vecspace#cosine".
```

3.3. Query Specification Examples

The query specifications for Examples 1 and 2 in Section 2 are coded as follows:

```
construct
    <Document>
  where
      <Document>
        <Article>
          <Date>
            <Year> $y </>
          </> element_as $d
          <Category/> content_as $c
          <Headline/> element_as $h
          <Body/>     element_as $b
        </> element_as $a
      </> in "articles.xml",
      $y > 2000
  order-by  $c
  construct
      <Group ID=CtgID($c)
             Category=$c>
        <Article>
           $d $h $a.abstract() $b
        </>
      </>
    </>
```

Query Specification 1

```
construct
    <Document>
  where
      <Document>
        <Article/> element_as $a
      </> in "articles.xml"
  rank-by
      sim_cosine($a,keywords) top N
  construct
      $a
    </>
```

Query Specification 2

Query Specification 1 extracts values from the XML instance in `articles.xml` and binds variables by the `where` clause. Variable `$a` is bound to each `Article` element in the root `Document` element. Variables `$d`, `$h` and `$b` are bound to `Date`, `Headline` and `Body` elements in `$a`, respectively. Variable `$c` is bound to the content of a `Category` element in `$a`. Variable `$y` is bound to the content of a `Year` element whose content is greater than 2000. The outer `construct` clause creates a `Document` whose content is the result of the subquery, which is a sequence of `Group` elements sorted by the `Category` values. Each `Group` element consists of `Article` elements with the same `Category` value. An `Article` element contains `Date`, `Headline`, `Summary` and `Body` elements. The `Summary` elements are created by the element method `abstract()` associated with the `Article`.

In Query Specification 2, the general function `sim_cosine()` is used. It gives the similarity of an `Article` element with respect to the given keywords. Using the `rank-by` clause, only the top N `Article` elements are contained in the result. They appear in the same order as in the source XML instance.

4. Extension Mechanism for Foreign Functions

Users who write foreign function implementations need to understand the foreign function framework. This framework is based on an XML data model that includes element objects, and the Java binding.

4.1. Data Model

XML data are generally composed from elements and `#PCDATA` text, and individual element types have their own attribute lists and content models. Each attribute consists of a name and value, which are strings. Content models are empty or sequences of elements and `#PCDATA` text.

In X^2QL, there are five built-in types: `string`, `number`, `boolean`, `content` and `element`; `string`, `content` and `element` are needed to express the above structure and the others are for convenience. Beyond that, all element types in XML instances can be used as types in X^2QL. They are treated as a subtype of the `element` type, so that `element` is a generic built-in type for any element in an XML instance. Element methods can be associated only with specific elements. The type hierarchy is shown in Figure 1.

When binding a variable, the value type is decided according to the location from which the value was extracted. The type is `string`, `content` or `element`. When evaluating an expression, the type of the resulting value is given by the type returned by the function.

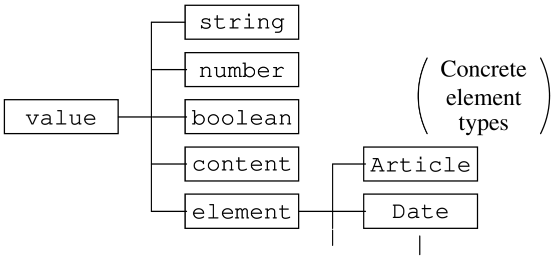

Figure 1. Type hierarchy.

When the type of a value differs from the type desired by a function in an expression, the value is translated into the desired type if possible. If translation is impossible, then the query is not processed. This translation is defined in Table 1. In this table, a function p_q $(p, q \in \{s, n, b, c, e\})$ maps a value of the type q into a value of the type p. Letters s, n, b, c and e represent `string`, `number`, `boolean`, `content` and `element`, respectively; "—" indicates that translation is unnecessary or undefined.

Functions s_c and s_e translate a content value and an element into a `string` value by removing tags within them, and s_n and s_b returns the string representation of a given value. When translating into `number` and `boolean` values, the values are first translated into a `string` value, then into `number` and `boolean` values by n_s and b_s. Functions c_s and c_e return a `content` value that contains only the given value. Only `content` values can be translated into `element` values. Function e_c returns the first element within a given `content` value if the first item is an element.

4.2. Internal Object Model

This subsection explains the internal abstract model for elements and their "dynamic aspects." The "dynamic aspect" of an element designates element methods and processing-time *properties* as shown in the ER diagram in Figure 2. Foreign function implementations are developed on the basis of this internal model. Language bindings such as the Java binding are actually used to implement them.

The associations between individual element types and implementations of element methods are given by foreign function definitions. These associations are managed by the *method table* for each element.

At the first invocation of an element method for a given element, an instance object called an *element object* is created for the element. All accesses to elements are made through element objects. The implementation is then executed through the corresponding function table, and the element object is passed to the implementation. Instantiated element objects go on living while processing a query.

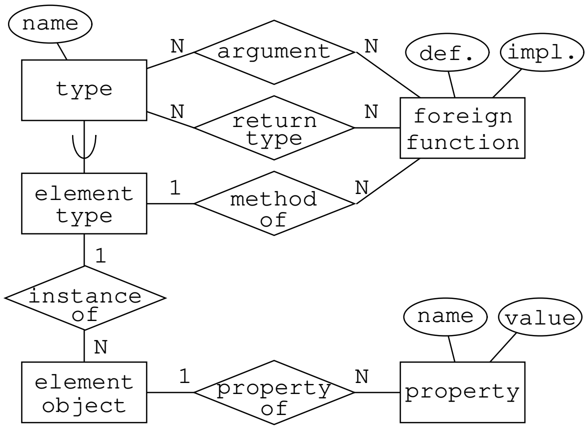

Figure 2. Internal abstract model.

When another element method for the same element is invoked, the element object that has already been instantiated is used. All general functions are managed by the anonymous element object.

Element objects have an extended interface of W3C DOM `Element` to provide accessibility to their static contents and dynamic aspects. Therefore, implementations can access the element itself, its parent, child nodes and the content within the element. They also serve functions to

- invoke an element method associated with the element,
- store data created by the implementation as properties.

The first feature is achieved by the `invoke()` method of element objects. This feature makes it possible for implementations to invoke element methods of other element objects. This invocation is enabled by an internal method table. The second feature is provided by the `setProperty()` and `getProperty()` methods. These data, stored as properties, are maintained in the element object during its lifetime. Therefore, they enable implementations of foreign functions whose results differ from or depend on the results of previous invocations. Properties are named by string keys and kept in a dictionary collection within the element object.

4.3. Java Binding

As mentioned earlier, the implementations of foreign functions are coded in Java programs. In Java binding, data types `string`, `number`, `boolean`, `content` and `element` are mapped into `XString`, `XNumber`, `XBoolean`, `XContent` and `XElement` interfaces in the package provided for X^{2}QL. All element types are treated as `element` values. These interfaces are subinterfaces of `XObject`.

Table 1. Translation rules between types

from \ to	string	number	boolean	content	element
string	—	$n_s(x)$	$b_s(x)$	$c_s(x)$	—
number	$s_n(x)$	—	$b_n(x)$	$c_s(s_n(x))$	—
boolean	$s_b(x)$	$n_b(x)$	—	$c_s(s_b(x))$	—
content	$s_c(x)$	$n_s(s_c(x))$	$b_s(s_c(x))$	—	$e_c(x)$
element	$s_e(x)$	$n_s(s_e(x))$	$b_s(s_e(x))$	$c_e(x)$	—

Note that `XContent` and `XElement` are extensions of the W3C DOM `DocumentFragment` and `Element` interfaces. `XContent` has no additional methods in the current version. `XElement` represents element objects. Additional methods of element objects explained in the previous subsection are defined in Java as follows:

```
package
  jp.ac.tsukuba.is.kde.x2ql.objects;

public interface XElement
       extends org.w3c.dom.Element {
  public XObject invoke(
       String name, XObject[] args );
  public void setProperty(
         String name, Object value );
  public Object getProperty(
         String name );
}.
```

When an element method is to be implemented, a Java class is created and the implementation is given as its method. When an element method is invoked for an element, the `XElement` object is added as the first argument `self` in the argument list of the Java function. As an example, the Java implementation for the method `abstract()` in Subsection 3.2 is listed below. The query processor checks the element type of the `XElement` object bound to `self` before the method invocation.

Implementations can create new values using a factory object that implements the `XObjectFactory` interface. The factory is provided by the `SystemModule` class.

```
// The class file should be
// located at http://fqdn/path/.
package pkg;
import jp.ac.tsukuba.is.kde
         .x2ql.objects.XElement;
import jp.ac.tsukuba.is.kde
         .x2ql.util.SystemModule;

public class ArticleMethods {
  public XElement abstract(
         XElement self ) {
    XElement result = null;
    ...
    result = SystemModule
            .getXObjectFactory()
            .createXElement( ... );
    return result;
  }
  ...
}
```

5. Other Features

This section discusses additional features of X^{2}QL. These features include aggregation functions and filtering functions.

5.1. Aggregation Functions

Support of a grouping facility by `group-by` clause and built-in aggregation functions is expected in a future version of XML-QL [4]. This subsection introduces user-defined aggregation functions, which calculate a value from a given set of values. Definitions are given by the following syntax, which is similar to the syntax for general functions.

`aggregation` *type-name*
aggregation-name `(` *arg-list* `)`
`defined-by` `"`*URI of the implementation*`"`.

The same Java binding as general functions is also provided.

Aggregation functions can be used in `construct` clauses. Optional `group-by` clauses can be used to group tuples. When a `group-by` clause is specified, tuples are grouped by argument values, then the aggregation function computes a value from each group of tuples. If the `group-by` clause is omitted, the aggregation function is applied to all tuples, so that the `construct` clause generates one output. Note that variables in a `group-by` clause must be used as arguments to the Skolem function in the outermost tag in the corresponding `construct` clause.

For queries including an aggregation function, tuples are first bound on the basis of *patterns* specified in `where` clauses. The aggregation function is called for each tuple.

The return values in these calls are ignored. When there are no more tuples, the aggregation function is called again with null argument values and its return value is used as the aggregated result.

For instance, a query counting the number of articles in each year is coded as follows:

```
aggregation number  count(
            element item )
defined-by  "http://fqdn/path/
             funcs.Counter#count"
```

and,

```
construct <Document>
  where     <Document>
              <Article>
                <Date>
                  <Year> $y </>
                </>
              </> element_as $a
            </> in "articles.xml"
  group-by  $y
  construct <Number ID=YearID ($y )
                    Year=$y>
             count( $a )
            </>
          </>.
```

The aggregation function `count()` is implemented as the following code. This Java class is instanciated for each group.

```
package funcs;
...
public class Counter {
  int number = 0;
      // The intermediate result
  public XNumber  count(
         XElement item ) {
    XNumber result = null;
    if( item != null )
      ++ number;
    else
      result = SystemModule
            .getXObjectFactory()
            .createXNumber( number );
    return result;
  }
}
```

5.2. Filtering Function

X^{2}QL provides the tuple filtering mechanism as another feature. This mechanism gets the sequence of tuples from its preceding clause, generates a new sequence of tuples from it, and passes the result to its following clause. This mechanism can be used in the `filter-by` clause. Filtering functions are used for this purpose. Their definition is given by the following syntax:

```
filter      filter-name( arg-list )
defined-by "URI of the implementation".
```

When a filtering function is evaluated in a query, a queue is given to output filtered tuples. The filtering function is called for each input tuple and argument values. It outputs tuples into the given queue based on the arguments. In this process, they are not allowed to modify the given tuples or to put new tuples into the output queue. When all the input tuples are consumed, an empty input tuple is given to the filtering function. In this call, the queue must be closed.

For example, a filtering function `head()` selecting the first `percent` % tuples is defined as follows:

```
filter     head( number percent )
defined-by "http://fqdn/path/
            funcs.FilterFunc#head".
```

A query to select the first twenty percent of the `Article` elements can be specified by using the filtering function `head()` as follows:

```
construct <Document>
  where     <Document>
              <Article/>
                  element_as $a
            </> in "articles.xml"
  filter-by head( 20 )
  construct $a
          </>.
```

The filtering function `head()` is implemented as the following code in Java binding.

```
package funcs;
...
public class FilterFunc {
  Queue spool = new Queue();
  int output = 0;
  int total = 0;

  public void       head(
         TupleQueue queue,
         Tuple      tuple,
         XNumber    percent ) {
    if( tuple != null ) {
      spool.push( tuple );
      ++ total;
    }
```

```
      flush( queue, percent );
      if( tuple == null )
        queue.close();
    }

    protected void       flush(
              TupleQueue queue,
              XNumber    percent ) {
      while( output <= total / 100.0
                 * percent.getValue() )
        queue.push((Tuple)spool.pop());
        ++ output
      }
    }
  }
```

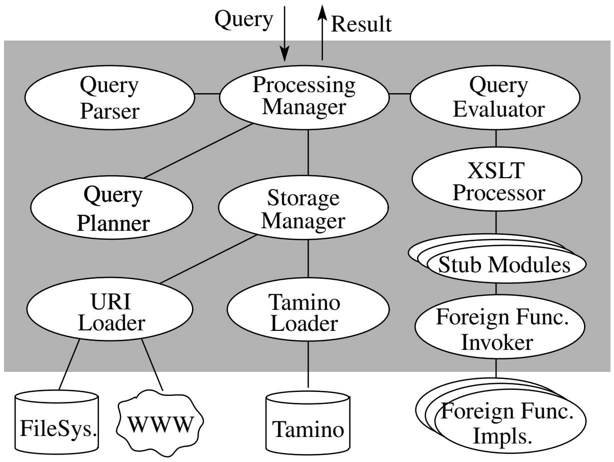

Figure 3. System archtecture.

6. Query Processing System

We developed a query processor that translates X^{2}QL queries into XSLT template rules and processes them using an XSLT processor [12]. It provides a basic query processing facility. We have been developing an X^{2}QL query processing system that can work with XML storages and process queries more efficiently [13]. This section describes development of the X^{2}QL query processing system.

6.1. System Architecture

Figure 3 shows the system architecture. This system consists of *Processing Manager*, *Query Parser*, *Query Planner*, *Storage Manager*, *XML Storages*, *Query Evaluator* and *Foreign Function Invoker*. Examples of XML Storages are XML servers, file systems and WWW sites. The current version can work with Tamino [15] to access XML instances. Tamino is an XML server that supports XQL.

The Processing Manager interacts with outer environments to get queries and return the results; it provides a simple UI for users and APIs for application programs. When given a query, this module processes it using other modules.

The Query Parser parses a given query, and then translates it into its *Query Information*, which is its internal representation. Query Information objects keep additional information about their variables, for example, their positions in structural conditions in `where` clauses, structural relationships among variables, and dependencies between variables and expressions in which they occur. This module has been developed by using JavaCC.

The Query Planner makes a *Processing Plan* for a given Query Information object. It decomposes the given Query Information object into partial conditions depending on individual data sources and the remaining core part integrating them. Those partial conditions are decided based on querying capability of the corresponding XML storages, then they are represented as *filtering queries*. For example, XQL filtering queries are generated to acquire XML instances managed by Tamino.

The Storage Manager acquires XML instances from each XML storage with the corresponding filtering query, file path or URI generated by the Query Planner. It supports pre-fetching and buffer management.

The Query Evaluator evaluates the core part of the query using an XSLT processor. It is based on the previous version of the X^{2}QL processor [12]. When the query includes a foreign function call, the Query Evaluator directly invokes the Java method implementing it if we can statically identify which foreign function is called. When we must dynamically identify it, for example, when a variable can be bound to more than two kinds of elements by wildcards, the Query Evaluator calls foreign functions of such element objects through the Foreign Function Invoker.

Stub Modules manage the difference between the data model and Java binding of X^{2}QL foreign functions and ones of XSLT extension functions. These modules are automatically generated for the corresponding element types by the Query Translator sub-module of the Query Evaluator. They have interfaces based on the Java binding of the XSLT processor own. They get Java `Method` objects representing foreign function implementation from the Foreign Function invoker, and delegate foreign function calls to them.

The Foreign Function Invoker checks the types of arguments and return values, and invokes foreign functions. It despatches the invocations to the appropriate element objects, then they also despatches the invocations to the foreign function implemantaions by looking up their own element method tables. Implementation classes are loaded by using `NetworkClassLoader` objects associated with individual base URIs. Note that classes sharing the same name but loadable from different URIs can be distinguished because of the Java specification.

6.2. Query Processing Flow

When a query is executed, the Processing Manager processes it in the following steps:

1. Gets the Query Information of the query from Query Parser.
2. Gets the Query Plan for the Query Information from Query Planner.
3. Gets XML instances from Storage Manager based on the Query Plan.
4. Processes the remaining core part of the query for the XML instances with the Query Evaluator.
5. Returns the query result.

7. Related Works

Several languages have been proposed to manipulate XML instances.

XML-QL [4] is a query language for XML documents, and it is used as a basis of X^2QL. In the current version of XML-QL, user-defined functions are merely parameterized queries. A proposal mentions possible use of user-defined predicates as a future research issue. However, their scope, role, syntax, and semantics are not specified. Foreign functions in X^2QL can also represent user-defined predicates. Beyond that, they can be used for a wide range of applications, and can return elements and contents.

Quilt [16] has emerged from designing experiments of XML query languages, and has powerful structure manipulation capability. Reports say that it can be extended by user-defined functions. Details of its extensibility, however, are not yet clear. Moreover, X^2QL provides more flexible extensibility. An example is the filtering functions. Note that Quilt was proposed after we started to develop X^2QL. XQuery [7], based on Quilt, has been proposed to W3C. The draft specification of XQuery mentions user-defined foreign functions as a future issue.

XSLT [17] is an XML transformation language. XSLT features powerful, tag-based data structure manipulation, but it requires low-level and procedural coding. The implementation framework of extension functions, which are user-defined functions in XSLT, depends on individual XSLT processors. The working draft of XSLT 1.1 [18] defines Java and ECMAScript bindings of extension functions.

XPath [5] and XQL [6] are simple XML path languages that have been proposed based on the pattern language in a previous version of XSL. They have no extensibility. They are widely used to express paths in more complex XML query languages, and are used as basic query languages in some XML servers.

YATL [8], StruQL [9], Lorel [10] and UnQL [11] are query languages for semistructured data, and XML data are their potential targets. YATL, in particular, allows user-defined external functions. However, YATL is not dedicated to XML data processing, and the use of foreign functions is limited to string-based pattern matching. The others have no extensibility.

XML query languages have different expressive power and underlying data models. To establish their foundation, the W3C Query Working Group has been discussing XML Query Requirements [19], XQuery and XPath Data Model [20] and XQuery Formal Semantics [21]. XQuery and XPath Data Model formally defines the information contained in the input to an XQuery processor. This is based on the XML Information Set [22], which provides a description of the information available in a well-formed XML instance. It also supports data types in XML Schema [23] [24] [25], which extends an instance of the XML Information Set with more precise information. Some features of X^2QL may not precisely coincide with these proposals. However, basic ideas and concepts to achieve the extensibility in X^2QL can be adopted even in their context.

8. Conclusion

Some XML query languages provide powerful, tag-based data structure manipulation. They are, however, generally weak in domain oriented content processing, such as summary generation and similarity calculation for news articles and feature extraction for images. Requirements for content processing vary with application context, so it is very difficult to provide a complete set of content processing facilities.

We have proposed X^2QL, which features the inclusion of user-defined foreign functions in the context of XML-QL-based XML querying facility. The extensibilty is based on an XML data model, in which elements are modeled as stateful objects. This paper explained the data model and the extension mechanisms. It also described development of an X^2QL query processor working with XML servers.

Future work includes inheritance of element methods and other object-oriented extensions. We are planning to make types in X^2QL compatible with those in standards, such as the XML Query Data Model and XML Schema.

Acknowledgment

This research was supported in part by the Grant-in-Aid for Scientific Research from Japan Society for the Promo-

tion of Science.

References

[1] D. Suciu and G. Vossen. proc. Third International Workshop on the Web and Databases (WebDB 2000), LNCS, Springer-Verlag, 2000.

[2] World Wide Web Consortium, http://www.w3.org/.

[3] M. Marchiori (ch.), The Query Languages Workshop (QL'98), http://www.w3.org/TandS/QL/QL98/, 1998.

[4] A. Deutsch and M. Fernandez and D. Florescu and A. Levy and D. Suciu. A Query Language for XML, Proc. the Eighth International World Wide Web Conference (WWW8), Computer Networks, vol. 31, no. 11–16, pp. 1155–1169, 1999.

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

[6] J. Robie and J. Lapp and and D. Schach. XML Query Language (XQL), The Query Languages Workshop (QL'98), http://www12.w3.org/TandS /QL/QL98/pp/xql.html, 1998.

[7] D. D. Chamberlin, J. Clark, D. Florescu, J. Robie, J. Simeón and M. Stefanescu, XQuery 1.0: An XML Query Language, http://www.w3.org/TR/xquery, W3C, 2001.

[8] S. Cluet and C. Delobel and J. Simeon and K. Smaga. Your Mediators Need Data Convention!, Proc. ACM-SIGMOD International Conference on Management of Data '98, pp. 177–188, 1998.

[9] M. F. Fernandez and D. Florescu and J. Kang and A. Y. Levy and D. Suciu. Catching the Boat with Strudel: Experiences with a Web-site Management System, Proc. ACM-SIGMOD International Conference on Management of Data '98, pp. 414–425, 1998.

[10] S. Abiteboul and D. Quass and J. McHugh and J. Widom and J. Wiener. The Lorel Query Language for Semistructured Data, International Journal on Digital Libraries, vol. 1, no. 1, pp. 68–88, 1997.

[11] P. Buneman and S. B. Davidson and G. G. Hillebrand and D. Suciu. A Query Language and Optimization Techniques for Unstructured Data, Proc. ACM-SIGMOD International Conference on Management of Data '96, pp. 506–516, 1996.

[12] N. Shinagawa, H. Kitagawa and Y. Ishikawa. X^2QL: An eXtensible XML Query Language Supporting User-defined Foreign Functions, Proc. 2000 ADBIS-DASFAA Symposium on Advances in Databases and Information Systems, LNCS 1884, Springer-Verlag, pp. 251–264, 2000.

[13] N. Shinagawa, T. Nagai, H. Kitagawa and Y. Ishikawa. Querying Geographic Data in XML via Extensible XML Query Language, Proc. International Symposium on Asia GIS 2001 (CD-ROM). 2001.

[14] G. Salton. Automatic Text Processing: The Transformation, Analysis and Retrieval of Information by Computer, Addison-Wesley, Reading, MA, 1989.

[15] Software AG, Tamino, http://www.softwareag.com/.

[16] D. D. Chamberlin, J. Robie and D. Florescu. Quilt: An XML Query Language for Heterogeneous Data Sources, Proc. Third International Workshop on the Web and Databases (WebDB 2000), pp. 53–62, 2000.

[17] J. Clark (ed.). XSL Transformations (XSLT) Version 1.0, http://www.w3.org/TR/xslt11, W3C, 1999.

[18] J. Clark (ed.). XSL Transformations (XSLT) Version 1.1, http://www.w3.org/TR/xslt11, W3C, 2001.

[19] P. Fankhauser, M. Marchiori and J. Robie (eds.). XML Query Requirements, http://www.w3.org/TR/xmlquery-req, W3C, 2001.

[20] M. Fernandez and J. Marsh (eds.). XQuery 1.0 and XPath 2.0 Data Model, http://www.w3.org/TR/query-datamodel, W3C, 2001.

[21] P. Fankhauser, M. Fernandez, A. Malhotra, M. Rys, J. Simeón and P. Wadler (eds.). XQuery 1.0 Formal Semantics, http://www.w3.org/TR/query-semantics/, W3C, 2001.

[22] J. Cowan and R. Tobin (eds.). XML Information Set, http://www.w3.org/TR/xml-infoset/, W3C, 2001.

[23] D. C. Fallside (ed.). XML Schema Part 0: Primer, http://www.w3.org/TR/xmlschema-0/, W3C, 2001.

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

[25] P. V. Biron and A. Malhotra. XML Schema Part 2: Datatypes, http://www.w3.org/TR/xmlschema-2/, W3C, 2001.