XQuark Bridge 1.0

XQuery Reference Guide

XQUARK BRIDGE 1.0

XQUERY REFERENCE GUIDE

Document version 1.0

FOREWORD

Status of the W3C references from which this document derives :

XML Schema Part 1: Structures. W3C Recommendation 2 May 2001. See http://www.w3.org/TR/2001/REC-xmlschema-1-20010502/

XML Schema Part 2: Datatypes. W3C Recommendation 2 May 2001. See http://www.w3.org/TR/2001/REC-xmlschema-2-20010502/

Namespaces in XML. W3C Recommendation 14 January 1999. See http://www.w3.org/TR/1999/REC-xml-names-19990114

XML Information Set. W3C Recommendation 24 October 2001. See http://www.w3.org/TR/2001/REC-xml-infoset-20011024

XML Path Language (XPath) version 1.0. W3C Recommendation 16 November 1999. See http://www.w3.org/TR/1999/REC-xpath-19991116

XQuery 1.0 An XML Query Language. W3C Working Draft 30 Avril 2002. See http://www.w3.org/TR/2002/WD-xquery-20020430

XQuery 1.0 and XPath 2.0 Functions and Operators. W3C Working Draft 30 April 2002. See http://www.w3.org/TR/2002/WD-xquery-operators-20020430/

XQuery 1.0 and XPath 2.0 Data Model. W3C Working Draft 30 April 2002. See http://www.w3.org/TR/2002/WD-query-datamodel-20020430/

XML Query Use Cases. W3C Working Draft 30 April 2002. See http://www.w3.org/TR/2002/WD-xmlquery-use-cases-20020430

XQuery 1.0 Formal Semantics. W3C Working Draft 26 March 2002. See http://www.w3.org/TR/2002/WD-query-semantics-20020326/

This section describes the status of these documents from W3C at the time of their publication. Other documents may supersede these documents. The latest status of these document series is maintained at the W3C.

Table of contents

Part

1

Abstract

XQuery is an XML query language designed by the W3C to be broadly applicable across a large variety of native or non-native XML data sources, including structured and semi-structured documents, relational databases, and object repositories. XQuark Bridge provides an implementation compatible with XQuery and applicable to relational data sources. It is a middleware that wraps a relational database into an XML view, which can then be queried using XQuery.

This document is the query reference guide for XQuark Bridge: it describes valid expressions of the language, as well as the specific, relational-backed XML data model to which the queries are applied.

Part 2

Introduction

Overview

XML has established itself as the standard data exchange format between applications on the Intranet and on the Internet. This has created the need for applications to publish their data in XML. As a large amount of today's business data is stored in relational databases, a general way of publishing relational data in XML is required. This requirement has been taken into consideration when designing the XQuery language: XQuery is an XML query language designed by the World-Wide Web Consortium[1] (W3C) to be broadly applicable across a large variety of native or non-native XML data sources, including structured and semi-structured documents, relational databases, and object repositories. XQuery is currently work in progress at the W3C. This user guide is based on the Working Draft dated April 30, 2002, which comprises four main documents [XQuery 1.0 An XML Query Language], [XQuery 1.0 and XPath 2.0 Functions and Operators], [XQuery 1.0 and XPath 2.0 Data Model] and [XML Query Use Cases].

XQuark Bridge provides an implementation compatible with XQuery and applicable to relational data sources. It does so by defining a generic XML view on top of a relational database schema, and by querying this view using XQuery expressions and built-in functions.

XQuark Bridge provides mechanisms to:

- Expose a subset of a relational schema as an XML database. This XML view exports its metadata information as a strongly-typed XML schema.
- Allow relational tables and views to be queried as collections of XML documents, using XQuery syntax.
- Execute strongly-types XQuery queries over the exposed XML collections.
- Return query results as newly constructed XML documents.

The above mechanisms represent a complete framework for efficiently publishing relational data in XML.

[1] The W3C is an organisation, widely supported by the industry, in charge of defining Internet-related standards, including XML and derived standards.

This reference guide is organized in four main sections:

- A description of the XML data model which is constructed by XQuark Bridge from the underlying relational model, and which defines the XML information available to the query processor.
- A description of the overall syntax used to express XQueries and their evaluation context.
- A guide to the XQuery language expressions, or more precisely to the subset of the XQuery expressions which is used by XQuark Bridge to query relational data.
- A list of available built-in functions.

Notations and conventions

This section introduces the typography used to present technical information in this manual.

The XQuark Bridge configuration files use a specific XML vocabulary to describe configuration options. In the XML representation, bold-face attribute names indicate a required attribute information item, and the rest are optional. Where an attribute information item has an enumerated type definition, the values are shown separated by vertical bars; if there is a default value, it is shown following a colon.

The allowed content of the information item is shown as a grammar fragment, using the Kleene operators `?` (0 or 1 occurrence), `*` (0 or more occurences) and + (1 or more occurrences).

```
<datasource
....name = xs:string>
....Content: (description?, url, user?, password?,
              substitutions?, catalog*)
</datasource>
```

The XML Schema that formally defines the XML vocabulary for configuration files is provided in Appendix A - XML Schema for the XQuark Bridge configuration file. This schema is associated to the namespace identified by the following URI: `http://www.xquark.org/Bridge/1.0/Datasource`.

XQuery expressions are described using grammar productions, based on a basic EBNF notation:

```
Query                ::= QueryProlog Expr
QueryProlog          ::= (NamespaceDecl
                         | DefaultNamespaceDecl)*
NamespaceDecl        ::= "namespace" NCName "="
                         StringLiteral
DefaultNamespaceDecl ::= "default element namespace ="
                         StringLiteral
```

Grammar productions within the body of the manual use only non-terminals, and all terminals are expanded for readability. Some basic non-terminals, defined in [XML Names] (e.g `QName` or `NCName`) are not defined in the manual body, but are present in the complete grammar for the XQuery language supported by XQuark Bridge, given in Appendix B – Complete BNF Grammar.

This document defines constructors and other functions that apply to one or more data types. Each constructor and function is defined by specifying its signature, a description of each of its arguments, and its semantics.

Each function's signature is presented in a form like this:

```
function-name(parameter-type $parameter-name,…)
=> return-type
```

In this notation, `function-name` is the name of the function whose signature is being specified. If the function takes no parameters, then the name is followed by an empty set of parentheses: (); otherwise, the name is followed by a parenthesized list of parameter declarations, each declaration specifying the static type of the parameter and a non-normative name used to reference the parameter when the function's semantics are specified. If there are two or more parameter declarations, they are separated by a comma. The `return-type` specifies the static type of the value returned by the function.

The function name is a `QName` and must adhere to its syntatic conventions. Following [XPath1.0], function names are composed of English words separated by hyphens,"-". If a function name contains a [XML Schema Part 2] datatype name, this may have intercapitalized spelling and is used in the function name as such. For example, `current-dateTime`. The functions discussed in this manual are contained in the namespace for built-in functions, namely `http://www.w3.org/2002/04/xquery-functions`. In XQuark Bridge, this namespace is the default namespace for function names, thus function names do not need to be prefixed.

As is customary, the parameter type name indicates that the function accepts arguments of that type in that position. If the parameter type name is one of the simple types defined in [XML Schema Part 2] the function also accepts

arguments with types derived from that type. These may be one of the derived types in [XML Schema Part 2] or they may be user-derived types.

Some functions accept the empty sequence as an argument and some may return the empty sequence. This is indicated in the function signature by following the parameter type name with a question mark:

```
function-name(parameter-type? $parameter-name) =>
return-type?
```

In this manual, the namespace prefixes `xs:` and `xsi:` are considered to be bound to the XML Schema namespaces `http://www.w3.org/2001/XMLSchema` and `http://www.w3.org/2001/XMLSchema-instance`, respectively. In some cases, where the meaning is clear and namespaces are not important to the discussion, built-in XML Schema typenames such as `integer` and `string` will be used without a namespace prefix.

Examples are provided throughout this manual as code listings, for instance:

```
for $u in collection("USERS")/USERS,
    $i in collection("ITEMS")/ITEMS
where $u/USERID = $i/OFFERED_BY
return
  <result>
    { $u/NAME }
    { $i/DESCRIPTION }
  </result>
```

Important notes, such as standard compliance notes, are presented as:

> **Note**: The JDBC type used when constructing the XML type represents the native type of the column in the database, not necessarily the one specified in the table creation statement. For instance, Oracle replaces all ANSI column type specifications by its own native types at table creation time.

References

[XML Schema Part 1] *XML Schema Part 1: Structures*. W3C Recommendation 2 May 2001. See http://www.w3.org/TR/2001/REC-xmlschema-1-20010502/

[XML Schema Part 2] *XML Schema Part 2: Datatypes*. W3C Recommendation 2 May 2001. See

http://www.w3.org/TR/2001/REC-xmlschema-2-20010502/

[XML Names] *Namespaces in XML*. W3C Recommendation 14 January 1999. See http://www.w3.org/TR/1999/REC-xml-names-19990114

[XML Infoset] *XML Information Set*. W3C Recommendation 24 October 2001. See http://www.w3.org/TR/2001/REC-xml-infoset-20011024

[XPath1.0] XML Path Language (XPath) version 1.0. W3C Recommendation 16 November 1999. See http://www.w3.org/TR/1999/REC-xpath-19991116

[XQuery 1.0 An XML Query Language] *XQuery 1.0 An XML Query Language*. W3C Working Draft 30 Avril 2002. See http://www.w3.org/TR/2002/WD-xquery-20020430

[XQuery 1.0 and XPath 2.0 Functions and Operators] *XQuery 1.0 and XPath 2.0 Functions and Operators*. W3C Working Draft 30 Avril 2002. See http://www.w3.org/TR/2002/WD-xquery-operators-20020430/

[XQuery 1.0 and XPath 2.0 Data Model] *XQuery 1.0 and XPath 2.0 Data Model*. W3C Working Draft 30 Avril 2002. See http://www.w3.org/TR/2002/WD-query-datamodel-20020430/

[XML Query Use Cases] *XML Query Use Cases*. W3C Working Draft 30 Avril 2002. See http://www.w3.org/TR/2002/WD-xmlquery-use-cases-20020430

[XQuery 1.0 Formal Semantics] *XQuery 1.0 Formal Semantics*. W3C Working Draft 26 March 2002. See http://www.w3.org/TR/2002/WD-query-semantics-20020326/

3

XQuery Data Model for XQuark Bridge

This section describes the XML view, defined on top of the relational database, which can be queried through XQuark Bridge.

The standard XQuery data model

XQuery introduces an XML data model, which defines precisely the information in an XML document that is available to an XQuery processor. It also defines all permissible values of expressions in the XQuery language.

The XQuery data model extends existing XML data models, such as the ones defined in XML Information Set [XML Infoset] or in XPath 1.0 [XPath1.0] by adding two new features to the model:

- Support for XML Schema types: XML elements, attributes and text nodes can be associated to structured complex types and simple data types, as defined in the XML Schema Recommendation (resp. [XML Schema Part 1] and [XML Schema Part 2]).
- Representation of collections of documents and complex elements.

Every value handled by the data model is either a *sequence* of zero or more *items*, or an *error*. An item is either a *node* or an *atomic value*.

A *node* is one of seven node kinds, as in the XPath 1.0 data model: document, element, attribute, namespace, processing-instruction, comment, text.

An *atomic value* encapsulates an XML Schema simple type and a corresponding value of that type.

A *sequence* is an ordered collection of nodes, atomic values, or any mixture of nodes and atomic values. A sequence cannot be a member of a sequence. A single item appearing on its own is modeled as a sequence containing one item.

The *error value* is a distinguished value used to identify error condions.

The XQuery data model can represent various values including not only the input and the output of a query, but all values of expressions used during

the intermediate calculations. Examples include the input document or document collection (represented as a document node or a sequence of document nodes), the result of a path expression (represented as a sequence of nodes), the result of an arithmetic or a logical expression (represented as an atomic value), a sequence expression resulting in a sequence of integers, dates, QNames or other XML Schema atomic values (represented as a sequence of atomic values), etc.

A complete specification of the XQuery data model can be obtained in the W3C draft recommendation [XQuery 1.0 and XPath 2.0 Data Model].

Typing in the standard XQuery data model

The XQuery data model relies on the XML Information Set obtained after XML Schema validity assessment. XML Schema validity assessment is the process of assessing an XML element information item with respect to an XML Schema and augmenting it and some or all of its descendants with properties that provide information about validity and type assignment. The result of schema validity assessment is an augmented Infoset, known as the Post Schema-Validation Infoset, or PSVI. The type information associated to each element node, attribute node or atomic value involves *schema components* of four different kinds: *element declaration*, *attribute declaration*, *complex type* and *simple type*, as defined in [XML Schema Part 1].

If validity has been successfully assessed, the item is guaranteed to be a valid instance of its associated type as defined by XML Schema. If not (either because no schema information was available or because the item is invalid), the item is associated with the permissive predefined XML Schema types xs:anyType (in the case of element nodes) or xs:anySimpleType (in the case of attribute nodes and atomic values).

Every node has a **typed value**, which is a sequence of atomic values. The typed value for the various kinds of nodes is defined as follows:

- The typed value of a document, namespace, comment, or processing instruction node is the error value.
- The typed value of a text node is the string content of the node, as an instance of `xs:anySimpleType`.
- The typed value of an element or attribute node that has no type annotation is a sequence of atomic values that is stored in the Data Model.
- The typed value of an element or attribute node whose type annotation denotes either a simple type or a complex type with simple content is a sequence of atomic values that is obtained by transforming the string content of the node into the value space of the associated type, as specified in [XML Schema Part 2].

- The typed value of an element node whose type annotation denotes a complex type with complex content is the error value.

The XQuark Bridge data model

The default XQuark Bridge data model is obtained by mapping relational structures and data into the XQuery data model introduced above. The result is an XML view where :

- The relational structures are exposed as one or several XML schemas, which contain all the *schema components* that are derived from the relational model. Those components are *element declarations*, which describe the XML structure of the content of each published relational table. By default, when accessing a single relational container[2], the generated schema does not have a target namespace. When accessing several containers, one XML schema is generated per container, and must necessarily be associated with a user-specified target namespace.
- Each exposed relational table is viewed as a named *collection of documents*, whose name is by default the name of the underlying relational object, optionally prefixed with the name of the relational container to which it belongs. Prefix and name are separated by a dot (e.g. ORDERS.CUSTOMER), as usual in relational databases.
- Each row in the relational table is viewed as a *document node*. The document has a top-level element which is schema-valid with respect to the element declaration generated from the table structure.

XQuark Bridge not only supports relational tables, but also views and synonyms, in a similar way. However, there are some limitations to the use of views and synonyms, as rows in those structures cannot be easily associated to identifiers. Those limitations appear when views or synonyms are used in nested queries, and are further detailed in the FLWR Expressions section of this manual.

XQuark Bridge also provides a way to control the XML view generation, by providing support for filtering and renaming tables and columns. This control is specified through a configuration file, whose syntax is detailed in Controlling the XML view generation.

Generated schema components

Each published table is associated to a generated element declaration, which provides an XML view of the table relational structure. The rules for generating the element declaration are listed below:

[2] A relational container is usually called a schema or a catalog, depending on the relational database vendor.

- The element declaration *name* is by default the name of the table, as internally represented in the database metadata. Unlike some relational databases, XML is case-sensitive, so a table name represented in upper case in the database will have to be used exclusively in upper case in queries. Table names that are not legal XML element names (e.g. those containing '$' or '#' signs) are ignored by XQuark Bridge, unless they are explicitly renamed in the XQuark Bridge configuration file, as detailed below. If the container of the table is associated to a target namespace, this namespace is the element declaration namespace, otherwise the element declaration does not have a namespace.
- The element declaration *type* is a complex type, whose content is a sequence containing a nested element declaration for each published column in the table.
- Each nested element declaration *name* is by default the name of the column, as internally represented in the database metadata. Column names that are not legal XML element names (e.g. those containing '$' or '#' signs) are ignored by XQuark Bridge, unless they are explicitly renamed in the XQuark Bridge configuration file, as detailed below. Nested element declarations are always considered local to their enclosing element, and therefore do not have a namespace.
- Each nested element declaration *type* is a predefined XML Schema simple type, obtained from the column JDBC type according to the table below. When a JDBC type is unsupported, the column is ignored (i.e. no nested element declaration will appear in the complex type).

JDBC Type	**XML type**
ARRAY	*not supported*
BIGINT	xs:long
BINARY	*not supported*
BIT	xs:boolean
BLOB	*not supported*
CHAR	xs:string
CLOB	*not supported*
DATE	xs:date
DECIMAL	xs:decimal
DISTINCT	*not supported*
DOUBLE	xs:double
FLOAT	xs:double
INTEGER	xs:integer
JAVA	*not supported*
LONGVARBINARY	*not supported*

LONGVARCHAR	*not supported*
NULL	*not supported*
NUMERIC	xs:decimal
OTHER	*not supported*
REAL	xs:float
REF	*not supported*
SMALLINT	xs:short
STRUCT	*not supported*
TIME	xs:time
TIMESTAMP	xs:dateTime
TINYINT	xs:byte
VARBINARY	*not supported*
VARCHAR	xs:string

Note: The JDBC type used when constructing the XML type represents the native type of the column in the database, not necessarily the one specified in the table creation statement. For instance, Oracle replaces all ANSI column type specifications by its own native types at table creation time.

Example

As an example, consider a relational database used by an online auction. The auction maintains a USERS table containing information on registered users, each identified by a unique userid, who can either offer items for sale or bid on items. An ITEMS table lists items currently or recently for sale, with the userid of the user who offered each item. A BIDS table contains all bids on record, keyed by the userid of the bidder and the item number of the item to which the bid applies.

The relational model for this example is defined below:

```
CREATE TABLE USERS (
  USERID        CHAR(3) PRIMARY KEY,
  NAME          VARCHAR(20) UNIQUE,
  RATING        CHAR(1)
);

CREATE TABLE ITEMS (
  ITEMNO        CHAR(4) PRIMARY KEY,
  DESCRIPTION   VARCHAR(30),
  OFFERED_BY    CHAR(3) REFERENCES USERS(USERID),
  START_DATE    DATE,
  END_DATE      DATE,
  RESERVE_PRICE NUMBER(10)
);
```

```
CREATE TABLE BIDS (
  USERID        CHAR(3) REFERENCES USERS(USERID),
  ITEMNO        CHAR(4) REFERENCES ITEMS(ITEMNO),
  BID           NUMBER(10) NOT NULL,
  BID DATE      DATE
);
```

The data for this example is given in the three tables below:

USERS		
USERID	NAME	RATING
U01	Tom Jones	B
U02	Mary Doe	A
U03	Dee Linquent	D
U04	Roger Smith	C
U05	Jack Sprat	B
U06	Rip Van Winkle	B

ITEMS					
ITEMNO	DESCRIPTION	OFFERED_BY	START_DATE	END_DATE	RESERVE_PRICE
1001	Red Bicycle	U01	99-01-05	99-01-20	40
1002	Motorcycle	U02	99-02-11	99-03-15	500
1003	Old Bicycle	U02	99-01-10	99-02-20	25
1004	Tricycle	U01	99-02-25	99-03-08	15
1005	Tennis Racket	U03	99-03-19	99-04-30	20
1006	Helicopter	U03	99-05-05	99-05-25	50000
1007	Racing Bicycle	U04	99-01-20	99-02-20	200
1008	Broken Bicycle	U01	99-02-05	99-03-06	25

BIDS			
USERID	ITEMNO	BID	BID_DATE
U02	1001	35	99-01-07
U04	1001	40	99-01-08
U02	1001	45	99-01-11
U04	1001	50	99-01-13
U02	1001	55	99-01-15
U01	1002	400	99-02-14
U02	1002	600	99-02-16
U03	1002	800	99-02-17
U04	1002	1000	99-02-25

BIDS			
USERID	ITEMNO	BID	BID_DATE
U02	1002	1200	99-03-02
U04	1003	15	99-01-22
U05	1003	20	99-02-03
U01	1004	40	99-03-05
U03	1007	175	99-01-25
U05	1007	200	99-02-08
U04	1007	225	99-02-12

The XML schema generated by XQuark Bridge[3] for this example is shown below:

```
<?xml version='1.0'?>

<schema xmlns="http://www.w3.org/2001/XMLSchema">

  <element name="BIDS">
    <complexType>
      <sequence>
 <element name="USERID" type="string"/>
 <element name="ITEMNO" type="string"/>
 <element name="BID" type="decimal"/>
 <element name="BID_DATE" type="dateTime"/>
      </sequence>
    </complexType>
  </element>

  <element name="ITEMS">
    <complexType>
      <sequence>
        <element name="ITEMNO" type="string"/>
        <element name="DESCRIPTION"
                 type="string"/>
        <element name="OFFERED_BY"
                 type="string"/>
        <element name="START_DATE"
                 type="dateTime"/>
        <element name="END_DATE"
                 type="dateTime"/>
        <element name="RESERVE_PRICE"
                 type="decimal"/>
      </sequence>
    </complexType>
  </element>

  <element name="USERS">
    <complexType>
      <sequence>
        <element name="USERID" type="string"/>
        <element name="NAME" type="string"/>
        <element name="RATING" type="string"/>
```

[3] This schema corresponds to the metadata information returned by the Oracle database for the relational model shown above. Other databases might create slightly different XML schemas.

```
      </sequence>
    </complexType>
  </element>
</schema>
```

Controlling the XML view generation

As described above, XQuark Bridge publishes a relational schema as a generic, strongly typed XML view, which can then be used as the basis for running XQueries. Although this generic approach is convenient in many situations, there are cases where finer control on the XML view generation is required. Those cases include:

- Applications which access several relational schemas,
- Applications which access only a small fraction of the relational tables in a relational schema,
- Applications which access tables and columns that are not legal XML element names.

For the benefit of those applications, XQuark Bridge provides configuration files that allow the application designer to better control the generated XML view. Configuration files are written in XML. Each file describes the wrapping of a single relational datasource, defined by the JDBC triple { JDBC URL, user, password }.

Selecting and filtering relational structures

The general structure of a configuration file is shown below:

```
<datasource name="{ Datasource identifier }">
  <description>
    { Optional datasource description }
  </description>
  <url> { JDBC connection string } </url>
  <user> { User name } </user>
  <password> { User password } </password>

  <substitutions>
    <nameCase>
      { lower | upper | mixed }
    </nameCase>
    <character value="{ character }"
               subst="{ substitution string }"/>
    …
  </substitutions>

  <catalog name="{ Optional catalog name }">
    <schema name="{ Optional schema name }"
            targetNamespace="{ Namespace URI }"
            elementFormDefault="{ qualified
                                | unqualified }">
```

```
      <includes>
        <table regex="{ Regular expression }"/>
        <table name="{ Table name }"
               alias="{ Table alias }">
          <includes>
            <column
              regex="{ Regular expression }"/>
            <column name="{ Column name }
                    alias="{ Column alias }"/>
          </includes>
          <excludes>
            <column
              regex="{ Regular expression }"/>
            <column name="{ Column name }"/>
          </excludes>
        </table>
      </includes>
      <excludes>
        <table regex="{ Regular expression }"/>
        <table name="{ Table name }" />
      </excludes>
    </schema>
  </catalog>
</datasource>
```

The complete XML Schema for the configuration file is given in Appendix A - XML Schema for the XQuark Bridge configuration file.

Three main sections appear in the configuration file:

- The datasource declaration,
- The substitutions declaration,
- The selection of the catalogs, schemas, tables and columns to be used in the XML view.

The datasource declaration section is composed of the following elements:

```
<datasource
....name = xs:string>
....Content: (description?, url, user?, password?,
              substitutions?, catalog*)
</datasource>

<description>
    Content: xs:string
</description>

<url>
    Content: xs:string
</url>

<user>
    Content: xs:string
</user>
```

```
<password>
    Content: xs:string
</password>
```

The role of the above elements and attributes is detailed below:

- The `name` attribute is an identifier defined by the application designer for this particular datasource.
- The optional `description` element is present for documentation purpose.
- The mandatory `url` element identifies the database instance to be wrapped.
- The optional `user` and `password` elements are used for the connection to the wrapped database instance.

The substitutions section is detailed in the next section.

The selection section is a set of hierarchical elements that represent traditional relational concepts:

```
<catalog
....name = xs:string>
....Content: schema+
</catalog>

<schema
....name = xs:string
    targetNamespace = xs:anyURI
    elementFormDefault = qualified | unqualified
                         : unqualified>
    Content: (includes?, excludes?)
</schema>

<includes>
    Content: table+ | column+
</includes>

<excludes>
    Content: table+ | column+
</excludes>

<table
    name = xs:string
    regex = xs:string
    alias = xs:NCName>
    Content: (includes?, excludes?)
</table>

<column
    name = xs:string
    regex = xs:string
    alias = xs:NCName />
```

The role of the above elements and attributes is detailed below:

- The `catalog` element can appear one or several times in the `datasource` top-level element, and represents a relational catalog in the wrapped database instance. The `name` attribute is optional and must be omitted if the database does not support the catalog concept: in this latter case, only a single, anonymous `catalog` element should appear in the configuration file. On the other hand, if more than one catalog are to be selected, each `catalog` element should have a `name` attribute, which represents the name of the catalog to be selected.

- The `schema` element can appear one or several times in a `catalog` element, and represents a relational schema in the enclosing catalog. The `name` attribute is optional and must be omitted if the database does not support the schema concept: in this latter case, only a single, anonymous `schema` element should appear in each `catalog` element. On the other hand, if more than one schema are to be selected, each `schema` element should have a `name` attribute, which represents the name of the schema to be selected. In addition, each `schema` element can have a `targetNamespace` attribute, which specifies the namespace of the generated element declarations associated to the schema tables. This attribute is optional only when XQuark Bridge accesses a single relational schema. In all other cases, each individual schema must be associated to a target namespace. When a target namespace is specified, an additional optional `elementFormDefault` attribute can be used to control the qualification of the inner generated element declarations (the ones corresponding to the table columns): if the attribute value is `qualified`, inner element declarations are qualified with the target namespace; if the attribute value is `unqualified`, or the attribute is absent, inner element declarations are not qualified.

- The `includes` element can appear zero or one time in a `schema` element (resp. a `table` element). It is used as a container for the elements that select tables (resp. columns) to be included in the generated XML view. When the `includes` element is absent, all tables (resp. columns) contained in the enclosing object are included.

- The `excludes` element can appear zero or one time in a `schema` element (resp. a `table` element). It is used as a container for the elements that select tables (resp. columns) to be excluded in the generated XML view.. Exclusion has higher priority than inclusion: a table which is both included and excluded will not appear in the XML view.

- The `table` element can appear one or several times in an `includes` or `excludes` element, and is used to select tables, views or synonyms[4] in the enclosing schema. One and only one of the `name` or `regex` attribute must be present. The `name` attribute selects the table of the given name in the enclosing schema: if no table corresponding to the name is found,

[4] In the following discussion, table is used to represent a relational table, view or catalog.

an error is generated. The `regex` attribute selects all the tables in the enclosing schema that have a name matching the specified regular expression. This regular expression uses the grammar described in [XML Schema Part 2], which is very close to the regular expression syntax of the Perl language. No error is generated if no match is found for the regular expression. When the `name` attribute is used, the element declaration generated for the table can be further refined by using an `includes` and/or `excludes` nested element to specify the columns to be used. This possibility is not available when the `regex` attribute is used. The use of the `alias` attribute is described in next section.

- The `column` element can appear one or several times in an `includes` or `excludes` element, and is used to select columns in the enclosing table. One and only one of the `name` or `regex` attribute must be present. The `name` attribute selects the column of the given name in the enclosing table: if no column corresponding to the name is found, an error is generated. The `regex` attribute selects all the columns in the enclosing table that have a name matching the specified regular expression. No error is generated if no match is found for the regular expression. The use of the `alias` attribute is described in next section.

Using aliases to rename relational structures

The XQuark Bridge configuration file also provides support for renaming relational structures. Renaming can be useful when:

- Table or column names contain characters that are not legal XML element names.
- Generated element declarations must match a predefined XML schema.

While both capabilities can be obtained using traditional relational database techniques such as views or synonyms, XQuark Bridge offers an additional level of flexibility through the configuration file.

Aliases for table and column names can be specified in two ways:

- Globally, by associating a substitution string to each unsupported character. XQuark Bridge will automatically substitute the string each time the character is encountered in a table or column name. The case of the generated table or column names can also be controlled globally.
- Locally, by associating an alias to a specific table or column.

Global substitutions are specified by adding an optional `substitutions` element after the datasource declaration section in the configuration file.

```
<substitutions>
    Content: (nameCase?, character*)
</substitutions>
```

```
<nameCase>
    Content: text() = mixed | upper | lower : mixed
</nameCase>

<character
    value = xs:string
    subst = xs:string />
```

This element contains:

- an optional `nameCase` element, which specifies the case management policy: *mixed* (the default) to preserve the case of the names returned by the database, *upper* or *lower* to impose a particular policy.
- one or several `character` elements, which specify the character to be replaced (the `value` attribute) and the substitution string (the `subst` attribute).

Local substitutions are specified by adding an optional `alias` attribute to the `table` or `column` element selecting the table or column to be renamed. The value of the attribute is the alias to be used by XQuark Bridge for the relational structure: the generated element declaration associated to the table or column will have the specified alias as name. The `alias` attribute can only be used in conjunction with the `name` attribute in `table` or `column` elements: structures selected through regular expressions cannot be renamed. Alias values are case-sensitive.

Part

4

The XQuery Prolog

The **Query Prolog** is a series of declarations that affect query processing. The Query Prolog is used to define namespace prefixes that are used in the query expression.

```
Query                ::= QueryProlog Expr
QueryProlog          ::= (NamespaceDecl
                         | DefaultNamespaceDecl)*
NamespaceDecl        ::= "namespace" NCName "="
                         StringLiteral
DefaultNamespaceDecl ::= "default element namespace ="
                         StringLiteral
```

A **namespace declaration** defines a namespace prefix and associates it with a namespace URI, adding the (prefix, URI) pair to the set of in-scope namespaces. The namespace URI must be a valid URI, and may not be an empty string. The namespace declaration is in scope for the rest of the query in which it is declared. Consider the following query:

```
namespace foo = "http://www.foo.com"
<foo:bar> Lentils </foo:bar>
```

In the query result, the newly created node is in the namespace associated with the namespace URI `http://www.foo.com`. The use of short prefixes as placeholders for longer namespace URIs is in line with the approach specified in [XML Names].

In XQuark Bridge, namespace URIs will either be:

- the namespaces associated to relational schemas in configuration files, or
- namespaces defining the structures of elements constructed in query results.

> **Compatibility note**: The XQuery draft standard allows schema declarations to be imported from schema files. This feature is not supported in XQuark Bridge, as all input data is associated to XML schemas generated from database structures.

In element constructors, namespace declaration attributes also associate a namespace with a prefix, adding a (prefix, URI) pair to the set of in-scope namespaces. In the data model, a namespace declaration is not an attribute, and it will not be retrieved by queries that return the attributes of an element. Namespace declarations are in scope within their containing element. Nested elements and attributes inherit the in-scope namespaces of their parents. The following query creates the same result as the previous query.

```
<foo:bar xmlns:foo="http://www.foo.com">
  Lentils
</foo:bar>
```

Because namespace declarations are in-scope within their containing element, they may be used in expressions that occur within an element constructor, as in the following query.

```
<foo:bar xmlns:foo="http://www.foo.com">
  { /foo:bing }
</foo:bar>
```

Names are compared on the basis of the expanded name (see [XML Names] for this and other namespace terms), not the QName. When element or attribute names are compared, they are considered identical if the local part and namespace URI match. Namespace prefixes are disregarded in name comparisons.

It is invalid to redefine already existing namespace prefixes.

```
{-- Error: attempt to redefine 'xx' --}
namespace xx = "http://www.foo.com"
namespace xx = "http://www.bar.com"
/xx:bing
```

It is also invalid to use a QName with a namespace prefix that has not been declared. The following query is also semantically invalid.

```
{-- Error: use of undeclared namespace prefix --}
/xx:bing
```

Namespace declaration attributes may redefine a namespace prefix within a given scope. The following query is valid.

```
namespace xx = "http://www.fe.com"
<xx:bar xmlns:xx = "http://www.fi.com">
  <xx:bing xmlns:xx = "http://www.fo.com">
    One
  </xx:bing>
  <xx:bing xmlns:xx = "http://www.fu.com">
    Two
  </xx:bing>
  <xx:bing> Three </xx:bing>
</xx:bar>
```

The result of the above query is as follows.

```
<xx:bar xmlns:xx = "http://www.fi.com">
  <xx:bing xmlns:xx = "http://www.fo.com">
    One
  </xx:bing>
  <xx:bing xmlns:xx = "http://www.fu.com">
    Two
  </xx:bing>
  <xx:bing> Three </xx:bing>
</xx:bar>
```

A **default namespace declaration** can be used to define the namespace URI to be associated with unprefixed element names. `default element namespace` defines a namespace URI that is associated with unprefixed names of elements.

> **Compatibility note**: The XQuery draft standard also defines a default namespace for unprefixed function names. In XQuark Bridge, this default namespace is always associated to the built-in XQuery functions URI, `http://www.w3.org/2002/04/xquery-operators`.

If no default element namespace is in effect, unqualified names of elements and types are in no namespace. Unqualified attribute names are always in no namespace, since XQuery provides no way to declare a default namespace for attributes.

The following example illustrates a default element namespace:

```
default element namespace = "http://www.foo.com"
<bar> Lentils </bar>
```

The result of the above query is shown below. Note that the name of the newly created element is in the namespace associated with the namespace URI `http://www.foo.com`, even though no namespace prefix occurs in the query.

```
<bar xmlns = "http://www.foo.com"> Lentils </bar>
```

Part

5

XQuery Expressions

Basics

The basic building block of XQuery is the **expression**. The language provides several kinds of expressions which may be constructed from keywords, symbols, and operands. In general, the operands of an expression are other expressions. XQuery is a **functional language** which allows various kinds of expressions to be nested. It is also a **strongly-typed language** in which the operands of various expressions, operators, and functions must conform to designated types. The following production defines valid top-level XQuery expressions:

```
Expr ::= PrimaryExpr
         | CommentExpr
         | PathExpr
         | AdditiveExpr
         | Constructor
         | FLWRExpr
         | SortExpr
```

The value of an expression is either a **sequence of items** (nodes or atomic values) belonging to the XQuery Data Model, or the special **error value**, which indicates that an error has been encountered during the evaluation of an expression. Except as noted in this document, if any operand of an expression is the error value, the value of the expression is also the error value.

Like XML, XQuery is a case-sensitive language. All keywords in XQuery use lower-case characters.

Expression Context

The **expression context** for a given expression consists of all the information that can affect the result of the expression. This information is organized into two categories called the **static context** and the **evaluation context**.

Static Context

The **static context** of an expression is defined as all information that is available during static analysis of the expression, prior to its evaluation. This

information can be used to decide whether the expression contains a static error.

In XQuery, the information in the static context is provided by declarations in the **query prolog** (except as noted below). Static context consists of the following components:

- **In-scope namespaces.** This is a set of (prefix, URI) pairs. The in-scope namespaces are used for resolving prefixes used in QNames within the expression.
- **Default namespace for element and type names.** This is a namespace URI. This namespace is used for any unprefixed QName appearing in a position where an element or type name is expected.
- **Default namespace for function names.** This is a namespace URI. This namespace is used for any unprefixed QName appearing as the function name in a function call. In XQuark Bridge, it is always bound to the namespace of the core XQuery functions and operators (`http://www.w3.org/2002/04/xquery-operators`).
- **In-scope variables.** This is a set of (QName, type) pairs. It defines the set of variables that have been declared and are available for reference within the XPath expression. The QName represents the name of the variable, and the type represents its static data type. Unlike the other parts of the static context, variable types are not declared in the query prolog. Instead, they are derived from static analysis of the expressions in which the variables are bound.

Evaluation Context

The **evaluation context** of an expression is defined as information that is available at the time the expression is evaluated. The evaluation context consists of all the components of the static context, and the additional components listed below.

The first component of the dynamic context (context item) is called the **focus** of the expression. The focus enables the processor to keep track of which nodes are being processed by the expression.

The focus for the outermost expression is supplied by the environment in which the expression is evaluated. Certain language constructs, notably the path expression E1/E2, the filter expression E1[E2], and the ordering expression E1 sortby E2, create a new focus for the evaluation of a sub-expression. In these constructs, E2 is evaluated once for each item in the sequence that results from evaluating E1. Each time E2 is evaluated, it is evaluated with a different focus. The focus for evaluating E2 is referred to below as the inner focus, while the focus for evaluating E1 is referred to as the outer focus. The inner focus exists only while E2 is being evaluated. When this evaluation is complete, evaluation of the containing expression continues with its original focus unchanged.

- The **context item** is the item currently being processed. An item is either an atomic value or a node. When the context item is a node, it can also be referred to as the **context node**. The context item is returned by the expression ".". When an expression `E1/E2`, `E1[E2]` or `E2 sortby E2` is evaluated, each item in the sequence obtained by evaluating `E1` becomes the context item in the inner focus for an evaluation of `E2`.
- **Dynamic variables.** This is a set of (QName, type, value) triples. It contains the same QNames as the in-scope variables in the static context for the expression. Each QName is associated with the dynamic type and value of the corresponding variable. The dynamic type associated with a variable may be more specific than the static type associated with the same variable. The value of a variable is, in general, a sequence. The dynamic types and values of variables are provided by execution of the XQuery expressions in which the variables are bound.
- **Current date and time.** This information represents a point in time during processing of a query. It can be retrieved by the `current-dateTime` function. If invoked multiple times during the execution of a query, this function always returns the same result.
- **Input sequence.** The input sequence is sequence of nodes that can be accessed by the `input` function. It might be thought of as an "implicit input". The content of the input sequence is determined in an implementation-dependent way: in the case of XQuark Bridge, it is the union of all collections (i.e. wrapped tables) available in the underlying relational database.

> **Compatibility note:** The XQuery draft standard defines three additional components in the focus, namely *context document*, *context position* and *context size*. Those components, inherited from the XPath1.0 standard, are not available in XQuark Bridge.

Input Functions

XQuery defines special functions that provide access to input data. These functions are of particular importance because they provide the only way in which an expression can reference a document or a collection of documents.

The **input sequence** is a part of the evaluation context for an expression. The `input` function returns the input sequence. For example, the expression `input()/customer` returns all the `customer` elements that are children of nodes in the input sequence. In XQuark Bridge, the input sequence is the union of all collections available in the underlying relational database. The `input` function can be omitted in path expressions: the expression `/customer` is equivalent to the expression `input()/customer`, and returns all the `customer` root elements that are present in the `customer` collection (i.e. all the rows present in the `customer` table).

The `collection` function returns the nodes found in a collection. In XQuark Bridge, a collection is backed by a relational table, and is viewed as a sequence of document nodes that represent rows in a relational table . A collection is identified by a string, which is the relational table name, optionally prefixed by the name of the relational container to which it belongs. For example, the expression `collection("customer")/customer` identifies all the `customer` root elements found in the collection which is backed by the `customer` relational table.

> **Compatibility note:** The XQuery draft standard defines an additional input function, called *document*. This function is not available in XQuark Bridge.

Expression Typing

XQuery is a strongly typed language with a type system based on [XML Schema Part 1]. The built-in types of XQuery include the node kinds of XML (such as element, attribute, and text nodes) and the built-in atomic types of [XML Schema Part 2] (such as `xs:integer` and `xs:string`). Additional complex types are defined by XQuark Bridge when wrapping one or several relational schemas, in the way described in Generated schema components.

When the type of a value is not appropriate for the context in which it is used, a **type exception** is raised. Any XQuery expression that raises a type exception returns the error value.

In XQuark Bridge, types are associated with values in one of the following ways:

- A literal value has a type; for example, the type of the value `47` is `xs:integer`.
- The constructor functions described in [XQuery 1.0 and XPath 2.0 Functions and Operators] return typed values; for example, `date("2002-05-31")` returns a value of type `xs:date`.
- When an instance of the Data Model is constructed from the database, it is associated to the generated XML type corresponding to the relational structure to which it belongs.
- Some functions, such as `data()`, extract typed values from nodes of the Data Model, preserving the types of these values.

Type Checking

XQuery provides two kinds of type checking, called **static type checking** and **dynamic type checking**.

Static type checking is performed during the query analysis phase (also known as "compile time.") Static type checking of an expression is based on the expression itself and on the **in-scope schema definitions**. Static type checking does not depend on the actual values found in any input document. The purpose of static type checking is to provide early detection of type errors and to compute the type of a query result.

During static type checking, each expression is assigned a static type. In some cases, the static type is derived from the lexical form of the expression; for example, the static type of the literal `5` is `xs:integer`. In other cases, the static type of an expression is inferred according to rules based on the static types of its operands; for example, the static type of the expression `size < 5` is `xs:boolean`.. The rules for inferring the static types of various expressions are described in [XQuery 1.0 Formal Semantics]. During the analysis phase, if an operand of an expression is found to have a static type that is not appropriate for that operand, a static error is raised. If static type checking raises no errors and assigns a static type T to an expression, then execution of the expression on valid input data is guaranteed to produce either a value of type T or the error value.

Dynamic type checking is performed during the query execution phase (also known as "run time.") Dynamic checking depends on the actual values found in input documents. At run time, a dynamic type is associated with each value as it is computed. The dynamic type of a value may be more specific than the static type of the expression that computed it (for example, the static type of an expression might be "zero or more integers or strings," but at run time its value may have the dynamic type "integer.") If an operand of an expression is found to have a dynamic type that is not appropriate for that operand, a type exception is raised.

It is possible for static type checking of an expression to raise a static type error, even though the expression might evaluate successfully on some valid input data. For example, an expression might contain a function that requires an element as its parameter, and static type checking might infer the static type of the function parameter to be an optional element. In this case, a static type error would result, even though the function call would be successful for input data in which the optional element is present.

It is also possible for an expression to return the error value, even though static type checking of the expression raised no error. For example, an expression may contain a constructor of an integer from a string, which is statically valid. However, if the actual value of the string at run time cannot be cast into an integer, the error value will result.

If an implementation can determine by static analysis that an expression will necessarily return the error value (for example, because it contains a division by the constant zero), the implementation is allowed to report this error at query analysis time (as well as at query execution time).

Type Conversions

Some expressions do not require their operands to exactly match the expected type. For example, function parameters expect a value of a particular type, but allow some basic conversions to be performed, such as extraction of atomic values from nodes, promotion of numeric values, and implicit casting of untyped values. Other operators that provide special conversion rules include arithmetic operators and value comparisons.

The following numerical type promotions are permitted:

- A value of type xs:decimal can be promoted to the type xs:float.
- A value of type xs:float can be promoted to the type xs:double.
- A value of a derived type can be promoted to its base type. As an example of this rule, a value of the derived type xs:integer can be promoted to its base type xs:decimal.

Type conversions sometimes depend on a process called **atomization**, which is used when an optional atomic value is expected. When atomization is applied to a given value, the result is either a single atomic value, an empty sequence, or a type exception. Atomization is defined as follows:

- If the value is a single atomic value or an empty sequence, atomization simply returns the value.
- If the value is a single node, the **typed value** of the node is extracted and returned; however, if the typed value is a sequence containing more than one item, a type exception is raised.
- In any other case, atomization raises a **type exception**.

Primary Expressions

Primary expressions are the basic primitives of the language. They include literals, variables, function calls, and the use of parentheses to control precedence of operators.

```
PrimaryExpr ::= Literal
                | FunctionCall
                | Variable
                | ParenthesizedExpr
```

Literals

A **literal** is a direct syntactic representation of an atomic value. XQuery supports two kinds of literals: string literals and numeric literals.

```
Literal        ::= NumericLiteral | StringLiteral
NumericLiteral ::= IntegerLiteral
                   | DecimalLiteral
                   | DoubleLiteral
IntegerLiteral ::= [0-9]+
DecimalLiteral ::= ("." [0-9]+) | ([0-9]+ "." [0-9]*)
DoubleLiteral  ::= (("." [0-9]+)
                    | ([0-9]+ ("." [0-9]*)?))
                   ([e] | [E]) ([+] | [-])? [0-9]+
StringLiteral  ::= (["] ([^"])* ["])
                   | (['] ([^'])* ['])
```

The value of a **string literal** is a singleton sequence containing an item whose primitive type is `xs:string` and whose value is the string denoted by the characters between the delimiting quotation marks.

The value of a **numeric literal** containing no "." and no `e` or `E` character is a singleton sequence containing an item whose type is `xs:integer` and whose value is obtained by parsing the numeric literal according to the rules of the `xs:integer` datatype. The value of a numeric literal containing "." but no `e` or `E` character is a singleton sequence containing an item whose primitive type is `xs:decimal` and whose value is obtained by parsing the numeric literal according to the rules of the `xs:decimal` datatype. The value of a numeric literal containing an `e` or `E` character is a singleton sequence containing an item whose primitive type is `xs:double` and whose value is obtained by parsing the numeric literal according to the rules of the `xs:double` datatype.

Here are some examples of literal expressions:

- `"12.5"` denotes the string containing the characters '1', '2', '.', and '5'.
- `12` denotes the integer value twelve.
- `12.5` denotes the decimal value twelve and one half.
- `125E2` denotes the double value twelve thousand, five hundred.

Values of other XML Schema built-in types can be constructed by calling the constructor for the given type. The constructors for XML Schema built-in types are defined in Built-in XQuery Functions. For example:

- `true()` and `false()` return the boolean values `true` and `false`, respectively.
- `integer("12")` returns the integer value twelve.
- `date("2001-08-25")` returns an item whose type is `xs:date` and whose value represents the date 25th August 2001.

Variables

A **variable** evaluates to the value to which the variable's NCName is bound in the **evaluation context**. If the variable's NCName is not bound, the value of the variable is the error value. Variables can be bound by clauses in **for expressions** and **quantified expressions**.

```
Variable ::= "$" NCName
```

Parenthesized Expressions

Parentheses may be used to enforce a particular evaluation order in expressions that contain multiple operators. For example, the expression `(2 + 4) * 5` evaluates to thirty, since the parenthesized expression `(2 + 4)` is evaluated first and its result is multiplied by five. Without parentheses, the expression `2 + 4 * 5` evaluates to twenty-two, because the multiplication operator has higher precedence than the addition operator.

```
ParenthesizedExpr ::= "(" Expr ")"
```

Function Calls

A **function call** consists of a QName followed by a parenthesized list of zero or more expressions. In XQuark Bridge, the QName must represent a built-in function. As XQuark Bridge automatically defines the built-in function namespace as the default namespace for functions, built-in function names can always be used without prefix. The expressions inside the parentheses provide the arguments of the function call. The number of arguments must equal the number of formal parameters in the function's signature; otherwise a static error is raised.

```
FunctionCall ::= QName "(" (Expr ("," Expr)*)? ")"
```

A function call expression is evaluated as follows:

- Each argument expression is evaluated, producing an argument value.
- Each argument value is converted to the declared type of the corresponding function parameter, using the function conversion rules listed below.
- The function is executed using the converted argument values. The result is a value of the function's declared return type.

The **function conversion rules** are used to convert an argument value or a return value to its required type; that is, to the declared type of the function parameter or return. The function conversion rules are as follows:

- If the required type is an atomic type:

- **Atomization** is applied to the given value. If the resulting atomic value is of type `xs:anySimpleType`, an attempt is made to cast it to the required type; if the cast fails, the function call returns the error value. If the atomic value has a type that can be promoted to the required type using the promotion rules described in Type Conversions, the promotion is done. After applying the above rules, if the resulting value does not conform to the required type, the function call returns the error value.

- If the required type is a sequence of items:
  - The given value is not converted. However, some functions may apply further additional conversion to their parameters: for instance, the `avg` function will attempt to convert any node in its input sequence into a numeric atomic value by getting its typed value. If this conversion fails, the function call returns the error value.

Comments

XQuery comments can be used to provide informative annotation. These comments are lexical constructs only, and do not affect the processing of an expression.

```
ExprComment ::= "{--" [^}]* "--}"
```

Comments may be used before and after major tokens within expressions and within element content..

Path Expressions

A **path expression** selects nodes within a tree or a sequence of trees (where a complex element node in the Data Model is seen as the root of a tree). A path expression is always evaluated with respect to an **evaluation context**.

```
PathExpr     ::= (PathExprRoot | StepExpr | PathExpr)?
                 "/" StepExpr
PathExprRoot ::= InputExpr | Variable
InputExpr    ::= "collection("  StringLiteral  ")"
                 | "input()"
```

> **Compatibility note**: Path expressions in XQuark Bridge compose a subset of the path expressions defined in the XQuery draft standard, which emcompasses and extends the XPath 1.0 expressions. As documents in XQuark Bridge have a very simple structure (featuring element nodes only with a depth of two), many features of the XQuery path expressions are not useful in XQuark Bridge. Examples include access to descendants at any hierarchical level (there is only one hierarchical level), access by position (all sequences of nodes are either of length 1 or unordered)…

A **path expression** consists of two expressions, separated by `/`. We will refer to the expression on the left side of / as E1 and the expression on the right side of / as E2. The expression E1 is evaluated, and if the result is not a sequence of nodes, the error value is returned. Each node resulting from the evaluation of E1 then serves in turn to provide an **inner focus** for an evaluation of E2. Each evaluation of E2 must result in a sequence of nodes; otherwise, the error value is returned. The sequences of nodes resulting from all the evaluations of E2 are merged, eliminating duplicate nodes based on identity and sorting the results in document order.

As an example of a path expression, `child::item/child::description` selects the `description` element children of the `item` element children of the context node, or, in other words, the `description` element grandchildren of the context node that have `div1` parents.

E1 expressions can be:

- Input functions: when a path expression is evaluated at the outermost level of a query (i.e. with an empty focus), E1 must be one of the two available input functions, `input` or `collection`. Both functions return a sequence of nodes which can be used as inner focus for the evaluation of E2.
- Variables: when a path expression is evaluated in an evaluation context in which some variables are bound, any variable bound to a node or a node sequence can be used as E1.
- Step expressions: when a path expression is evaluated in an evaluation context in which a focus is defined, a step expression can be used as E1.
- Path expressions: several path expressions can be concatenated, the result of the leftmost expressions being used recursively as the evaluation context for the following one.
- Absent: when a path expression starts with a "/" (i.e. E1 is absent), the **input sequence** is used as the sequence of nodes used as inner focus for the evaluation of E2.

E2 expressions are always step expressions, which are further described below.

Steps

```
StepExpr   ::= Step Predicates
Step       ::= (Axis NodeTest) | AbbreviatedStep
```

A **step** is an expression that returns a sequence of nodes, in document order and without duplicates. Steps are often used inside path expressions, and must always be evaluated in an evaluation context in which the focus is defined. A step might be thought of as beginning at the context node, navigating to those nodes that are reachable from the context node via a predefined axis, and selecting some subset of the reachable nodes. A step has three parts:

- an **axis**, which specifies the relationship between the nodes selected by the step and the context node. The axis might be thought of as the "direction of movement" of the step.
- a **node test**, which specifies the node kind and/or name of the nodes selected by the step.
- zero or more **predicates**, which further modify the sequence of nodes selected by the step.

In the **abbreviated syntax** for a step, the axis can be omitted and other shorthand notations can be used.

The unabbreviated syntax for an step consists of the axis name and node test separated by a double colon, followed by zero or more **predicates**. For example, in `child::para[child::title = "Introduction"]`, `child` is the name of the axis, `para` is the node test and `[child::title = "Introduction"]` is a predicate.

The node sequence selected by a step is found by generating an initial node sequence from the axis and node test, and then applying each of the qualifiers in turn. The initial node sequence consists of the nodes reachable from the context node via the specified axis that have the node kind and/or name specified by the node test. For example, the step `child::para` selects the `para` element children of the context node: `child` specifies that each node in the initial node sequence must be a child of the context node, and `para` specifies that each node in the initial node sequence must be an element named `para`.

Axes

```
Axis  ::= "child" "::"
        | "attribute" "::"
        | "self" "::"
```

XQuark Bridge supports three axes:

- the `child` axis contains the children of the context node.
- the `attribute` axis contains the attributes of the context node; the axis will be empty unless the context node is an element.
- the `self` axis contains the context node itself.

> **Compatibility note:** The XQuery draft standard defines several additional axes: descendant, descendant-or-self and parent. Those axes are not necessary in XQuark Bridge, due to the simple structure of the documents in the Data Model.

Node Tests

A **node test** is a condition that must be true for each node selected by a step. The condition may be based on the kind of the node (element, attribute, text, document, comment, processing instruction, or namespace) or on the name of the node.

```
NodeTest ::=   KindTest | NameTest
NameTest ::=   QName
KindTest ::=   "text" "(" ")"
               "node" "(" ")"
```

Every axis has a **principal node kind**. For both the `child` and `self` axes, the principal node kind is element. For the `attribute` axis, the principal node kind is attribute.

A node test that is a QName is true if and only if the **kind** of the node is the principal node kind and the expanded-name of the node is equal to the expanded-name specified by the QName. For example, `child::para` selects the `para` element children of the context node; if the context node has no `para` children, it selects an empty set of nodes. `attribute::href` selects the `href` attribute of the context node; if the context node has no `href` attribute, it selects an empty set of nodes. Note that static type checking is enforced: if the compiler can derive from the context node type that the name test will always be false, it will throw a type exception at compile time.

A QName in a node test is expanded into an expanded-name using the **in-scope namespaces** in the expression context. An unprefixed QName used as a nametest has the namespaceURI associated with the default element

namespace in the expression context. It has no namespace if the default element namespace is not defined in the expression context. It is an error if the QName has a prefix that does not correspond to any in-scope namespace.

The node test `text()` is true for any text node. For example, `child::text()` will select the text node children of the context node.

A node test `node()` is true for any node whatsoever.

Predicates

A predicate consists of an expression, called a **predicate expression**, enclosed in square brackets. A predicate serves to filter a node sequence, retaining some nodes and discarding others. For each node in the node sequence to be filtered, the predicate expression is evaluated using an **inner focus** derived from that node. The result of the predicate expression is coerced to a Boolean value, called the **predicate truth value**, as described below. Those nodes for which the predicate truth value is `true` are retained, and those for which the predicate truth value is `false` are discarded.

```
Predicates ::=   ("[" OrExpr "]")*
```

The predicate truth value is derived by applying the following rules, in order:

- If the value of the predicate expression is an empty sequence, the predicate truth value is `false`.
- If the value of the predicate expression is an atomic value of type `xs:boolean`, the predicate truth value is equal to the value of the predicate expression.
- If the value of the predicate expression is a sequence that contains at least one node and does not contain any item that is not a node, the predicate truth value is `true`. The predicate truth value in this case does not depend on the content of the node(s).
- In any other case, a type exception is raised.

Here are some examples of steps that contain predicates:

- This example selects all the children of the context node whose name is "toy" and whose "color" attribute has the value "red":

```
child::toy[attribute::color = "red"]
```

- This example selects all the "employee" children of the context node that have a "secretary" subelement:

```
child::employee[child::secretary]
```

Note that the above rules imply that `child::person[child:married]` returns all `person` children of the context node that have a `married` subelement, even if the content of this subelement is the boolean value *false*. In order to return married persons, the expression should be written:

```
child::person[data(child:married) = true()]
```

Unabbreviated Syntax

This section provides a number of examples of path expressions in which the axis is explicitly specified in each step. The syntax used in these examples is called the **unabbreviated syntax**. In many common cases, it is possible to write path expressions more concisely using an **abbreviated syntax**.

- `child::para` selects the `para` element children of the context node
- `child::text()` selects all text node children of the context node
- `child::node()` selects all the children of the context node, whatever their node type
- `attribute::name` selects the `name` attribute of the context node
- `self::para` selects the context node if it is a `para` element, and otherwise selects nothing
- `child::chapter/child::para` selects the `para` element children of the `chapter` element children of the context node
- `child::chapter[child::title='Introduction']`selects the `chapter` children of the context node that have one or more `title` children with string-value equal to `Introduction`
- `child::chapter[child::title]` selects the `chapter` children of the context node that have one or more `title` children
- `child::node()[self::chapter or self::appendix]` selects the `chapter` and `appendix` children of the context node

Abbreviated Syntax

```
AbbreviatedStep ::=   "." | (@ NameTest) | NodeTest
```

The abbreviated syntax permits the following abbreviations:

- The most important abbreviation is that `child::` can be omitted from a step. In effect, `child` is the default axis. For example, a path expression `section/para` is short for `child::section/child::para`.
- There is also an abbreviation for attributes: `attribute::` can be abbreviated by `@`. For example, a path expression `para[@type="warning"]` is short for

`child::para[attribute::type="warning"]` and so selects `para` children with a `type` attribute with value equal to `warning`.

- A step consisting of `.` is short for `self::node()`.

Here are some examples of path expressions that use the abbreviated syntax:

- `para` selects the `para` element children of the context node
- `text()` selects all text node children of the context node
- `@name` selects the `name` attribute of the context node
- `.` selects the context node
- `chapter[title="Introduction"]` selects the `chapter` children of the context node that have one or more `title` children with string-value equal to `Introduction`
- `chapter[title]` selects the `chapter` children of the context node that have one or more `title` children
- `employee[secretary and assistant]` selects all the `employee` children of the context node that have both a `secretary` subelement and an `assistant` subelement.

Arithmetic Expressions

XQuery provides arithmetic operators for addition, subtraction, multiplication, division, and modulus, in their usual binary and unary forms. Usual precedence rules apply.

```
AdditiveExpr       ::= (AdditiveExpr ("+"  |  "-"))?
                       MultiplicativeExpr
MultiplicativeExpr ::= (MultiplicativeExpr
                        ("*" | "div" | "mod"))?
                       UnaryExpr
UnaryExpr          ::= ("-" | "+")?
                       (PrimaryExpr
                        | PathExpr
                        | StepExpr)
```

The binary subtraction operator must be preceded by white space if it follows an NCName, in order to distinguish it from a hyphen, which is a valid name character. For example, `a-b` will be interpreted as a single token.

An arithmetic expression is evaluated by applying the following rules, in order, until an error is encountered or a value is computed:

- **Atomization** is applied to each operand, resulting in a single atomic value or an empty sequence for each operand.

- If either operand is an empty sequence, the result of the operation is an empty sequence.
- If an operand has the type `xs:anySimpleType`, it is cast to `xs:double`. If the cast fails, the error value is returned.
- If the two operands have different types, and these types can be promoted to a common type using the standard promotion rules, the operands are both promoted to their least common type. For example, if the first operand is of type `hatsize` which is derived from `xs:decimal`, and the second operand is of type `shoesize` which is derived from `xs:integer`, then both operands are promoted to the type `xs:decimal`.
- If the operand type(s) are valid for the given operator, the operator is applied to the operand(s), resulting in an atomic value or an error (for example, an error might result from dividing by zero.). If the operand type(s) are not valid for the given operator, a type exception is raised.

In XQuark Bridge, arithmetic operators are only supported for numeric types, thus arithmetic operations always result in numeric values. Static type checking is enforced: using an expression returning a sequence as an operand of an arithmetic operation will generate a type expression, unless the compiler can determine at compile time that the sequence will be of length 0 or 1.

Here are some examples of arithmetic expressions:

- Arithmetic operations on numeric values result in numeric values:

```
($salary + $bonus) div 12
```

- This example illustrates the difference between a subtraction operator and a hyphen:

```
$unit-price - $unit-discount
```

- Unary operators have higher precedence than binary operators, subject of course to the use of parentheses:

```
-($bellcost + $whistlecost)
```

> **Compatibility note:** The XQuery draft standard defines arithmetic operations on dates and durations. Those operations are not supported in XQuark Bridge.

Comparison Expressions

Comparison expressions allow two values to be compared.

```
CompExpr ::= AdditiveExpr
             ("=" | "!=" | "<" S | "<=" | ">" | ">=")
             AdditiveExpr
```

The "<" comparison operator must be followed by white space in order to distinguish it from a tag-open character

Comparisons can involve single values or sequences. In the absence of errors, the result of a comparison is always true or false. When the two operands are single values, the result of the comparison is defined by applying the following rules, in order:

- **Atomization** is applied to each operand, resulting in a single atomic value or an empty sequence for each operand.
- If either operand is an empty sequence, the result is an empty sequence.
- If either operand has the type `xs:anySimpleType`, that operand is cast to a required type, which is determined as follows:
  - If the type of the other operand is numeric, the required type is `xs:double`.
  - If the type of the other operand is `xs:anySimpleType`, the required type is `xs:string`.
  - Otherwise, the required type is the type of the other operand.

  If the cast fails, the error value is returned.
- If the comparison has two numeric operands of different types, one of the operands is promoted to the type of the other operand, following the promotion rules. For example, a value of type `xs:integer` can be promoted to `xs:decimal`, and a value of type `xs:decimal` can be promoted to `xs:double`.
- The result of the comparison is `true` if the value of the first operand is (equal, not equal, less than, less than or equal, greater than, greater than or equal) to the value of the second operand; otherwise the result of the comparison is `false`. If the value of the first operand is not comparable with the value of the second operand, a type exception is raised. XQuark Bridge allows strings, numeric values and dates to be compared to values of the same type.

Sequence comparisons are defined by adding existential semantics to single value comparisons. The operands may be sequences of any length greater than 1.

The comparison `A = B` is `true` for sequences `A` and `B` if the value comparison `a = b` is `true` for some item `a` in `A` and some item `b` in `B`. Otherwise, `A = B` is `false`.

Similarly:

- `A != B` is `true` if and only if `a != b` is `true` for some `a` in `A` and some `b` in `B`.
- `A < B` is `true` if and only if `a < b` is `true` for some `a` in `A` and some `b` in `B`.
- `A <= B` is `true` if and only if `a <= b` is `true` for some `a` in `A` and some `b` in `B`.
- `A > B` is `true` if and only if `a > b` is `true` for some `a` in `A` and some `b` in `B`.
- `A >= B` is `true` if and only if `a >= b` is `true` for some `a` in `A` and some `b` in `B`.

The sequence comparison may result in the error value if the value comparison of any two values from A and B results in the error value.

Logical Expressions

A **logical expression** is either an **and-expression** or an **or-expression**. In the absence of errors, the value of a logical expression is always one of the boolean values `true` or `false`.

```
OrExpr   ::= (OrExpr "or")? AndExpr
AndExpr  ::= (AndExpr "and")? BoolExpr
BoolExpr ::= CompExpr
             | PrimaryExpr
             | PathExpr
             | StepExpr
             | QuantifiedExpr
```

The first step in evaluating a logical expression is to reduce each of its operands to an **effective boolean value**, which is `true`, `false`, or the error value. The effective boolean value of an operand is defined as follows:

- If the operand is an empty sequence, its effective boolean value is `false`.
- If the operand is an atomic value of type `xs:boolean`, the operand serves as its own effective boolean value.
- If the operand is a sequence that contains at least one node and does not contain any item that is not a node, its effective boolean value is `true`.
- In any other case, a type exception is raised. In XQuery, a type exception always results in the error value.

The value of an and-expression is determined by the effective boolean values (EBV's) of its operands, according to the following table:

	EBV_2 = true	EBV_2 = false	EBV_2 = error
EBV_1 = true	true	false	error
EBV_1 = false	false	false	false or error
EBV_1 = error	error	false or error	error

The value of an or-expression is determined by the effective boolean values (EBV's) of its operands, according to the following table:

	EBV_2 = true	EBV_2 = false	EBV_2 = error
EBV_1 = true	true	true	true or error
EBV_1 = false	true	false	error
EBV_1 = error	true or error	error	error

The order in which the operands of a logical expression are evaluated is not deterministic. The tables above are defined in such a way that an or-expression can return `true` if the first expression evaluated is true, and it can return the error value if the first expression evaluated contains an error. Similarly, an and-expression can return `false` if the first expression evaluated is false, and it can return the error value if the first expression evaluated contains an error. As a result of these rules, the value of a logical expression is not deterministic in the presence of errors, as illustrated in the examples below.

Here are some examples of logical expressions:

- The following expressions return `true`:

```
1 = 1 and 2 = 2
1 = 1 or 2 = 3
```

- The following expression may return either `false` or the error value:

```
1 = 2 and 3 div 0 = 47
```

- The following expression may return either `true` or the error value:

```
1 = 1 or 3 div 0 = 47
```

- The following expression returns the error value:

```
1 = 1 and 3 div 0 = 47
```

In addition to and- and or-expressions, XQuery provides a function named `not` that takes a general sequence as parameter and returns a boolean value. The `not` function reduces its parameter to an effective boolean value using the same rules that are used for the operands of logical expressions. It then returns `true` if the effective boolean value of its parameter is `false`, and

`false` if the effective boolean value of its parameter is `true`. If the effective boolean value of its operand is the error value, `not` returns the error value.

Constructors

XQuery provides constructors that can create XML structures within a query. There are constructors for elements, attributes, CDATA sections, processing instructions, and comments.

```
Constructor          ::= ElementConstructor
                         | XmlComment
                         | XmlProcessingInstruction
                         | CdataSection
ElementConstructor   ::= "<" QName AttributeList
                         ("/>" | (">" ElementContent*
                         "</" QName ">"))
ElementContent       ::= Char
                         | "{{"
                         | "}}"
                         | ElementConstructor
                         | EnclosedExpr
                         | CdataSection
                         | CharRef
                         | PredefinedEntityRef
                         | XmlComment
                         | XmlProcessingInstruction
AttributeList        ::= (QName "=" AttributeValue)*
AttributeValue       ::= (["]
                          ("'" | AttrValueContent)*
                          ["])
                         |(['])
                           (""" | AttrValueContent)*
                           ['])
AttrValueContent     ::= Char
                         | CharRef
                         | "{{"
                         | "}}"
                         | EnclosedExpr
                         | PredefinedEntityRef
EnclosedExpr         ::= "{" Expr "}"
Char                 ::= [#x0009] | [#x000D]
                         | [#x000A] | [#x0020-#xFFFD])
CharRef              ::= "&#" ([0-9]+
                         | ("x"([0-9] | [a-f]
                            | [A-F])+)) ";"
PredefinedEntityRef ::= "&" ("lt" | "gt" | "amp"
                         | "quot" | "apos") ";"
```

Element Constructors

An **element constructor** creates an XML element. If the name, attributes, and content of the element are all constants, the element constructor uses standard XML notation. For example, the following expression creates a `book` element that contains attributes, subelements, and text:

```
<book isbn="isbn-0060229357">
  <title>Harold and the Purple Crayon</title>
  <author>
    <first>Crockett</first>
    <last>Johnson</last>
  </author>
</book>
```

In an element constructor, the name used in an end tag must match the name of the corresponding start tag. If **namespace prefixes** are declared in the **query prolog**, the prefixes they declare may be used to create **qualified names** for elements and attributes. It is an error to use a **namespace prefix** that has not been declared.

In an element constructor, curly braces { } delimit **enclosed expressions**, distinguishing them from literal text. Enclosed expressions are evaluated and replaced by their value, whereas material outside curly braces is simply treated as literal text, as illustrated by the following example:

```
<example>
  <p> Here is a query. </p>
  <eg> $i//title </eg>
  <p> Here is the result of the above query. </p>
  <eg>{ $i//title }</eg>
</example>
```

The above query might generate the following result (whitespace has been added for readability to this result and other result examples in this document):

```
<example>
  <p> Here is a query. </p>
  <eg> $i//title </eg>
  <p> Here is the result of the above query. </p>
  <eg>
    <title>Harold and the Purple Crayon</title>
  </eg>
</example>
```

In an element constructor, an enclosed expression may evaluate to any sequence of nodes and/or atomic values. Attribute nodes occurring in this sequence become the attributes of the constructed element. The remainder of the sequence becomes the content of the constructed element.

An enclosed expression may also be used to compute the value of an attribute. If the enclosed expression returns a node, the **typed value** of the

node is extracted and assigned to the attribute, as illustrated by the following example:

```
<book isbn="{$i/@booknum}" />
```

Since XQuery uses curly braces to denote enclosed expressions, some convention is needed to denote a curly brace used as an ordinary character. For this purpose, XQuery adopts the same convention as XSLT: Two adjacent curly braces in an XQuery character string are interpreted as a single curly brace character.

Other Constructors and Comments

The syntax for a **CDATA section constructor**, a **processing instruction constructor**, or an **XML comment constructor** is the same as the syntax of the equivalent XML construct.

```
CdataSection             ::= "<![CDATA[" Char* "]]>"
XmlProcessingInstruction ::= "<?" NCName Char* "?>"
XmlComment               ::= "<!--" Char* "-->"
```

The following example illustrates constructors for processing instructions, comments, and CDATA sections.

```
<?format role="output" ?>
<!-- Tags are ignored in the CDATA section -->
<![CDATA[
  <address>
     123 Roosevelt Ave. Flushing, NY 11368
  </address>
]]>
```

Note that an **XML comment** actually constructs an XML comment node. An XQuery comment is simply a comment used in documenting a query, and is not evaluated. Consider the following example.

```
{-- This is an XQuery comment --}
<!-- This is an XML comment -->
```

The result of evaluating the above expression is as follows.

```
<!-- This is an XML comment -->
```

FLWR Expressions

XQuery provides a FLWR expression for iteration and for binding variables to intermediate results. This kind of expression is often useful for computing joins between two or more documents or collections of documents and for restructuring data. The name "FLWR", pronounced

"flower", stands for the keywords `for`, `let`, `where`, and `return`, the four clauses found in a FLWR expression.

```
FLWRExpr    ::= (ForClause | LetClause)+
                WhereClause? "return"  Expr
ForClause   ::= "for" Variable "in" Expr
                ("," Variable "in" Expr)*
LetClause   ::= "let" Variable ":=" Expr
                ("," Variable ":=" Expr)*
WhereClause ::= "where" OrExpr
```

The clauses of a FLWR Expression are interpreted as follows:

- A `for` clause associates one or more variables with expressions, creating tuples of variable bindings drawn from the Cartesian product of the sequences of values to which the expressions evaluate. The variable binding tuples are generated as an ordered sequence as described below.
- A `let` clause binds a variable directly to an entire expression. If `for` clauses are present, the variable bindings created by `let` clauses are added to the tuples generated by the `for` clauses. If there are no `for` clauses, the `let` clauses generate one tuple with all variable bindings.
- A `where` clause can be used as a filter for the tuples of variable bindings generated by the `for` and `let` clauses. The expression in the `where` clause, called the **where-expression**, is evaluated once for each of these tuples. If the **effective boolean value** of the where-expression is `true`, the tuple is retained and its variable bindings are used in an execution of the `return` clause. If the **effective boolean value** of the where-expression is `false`, the tuple is discarded.
- The `return` clause contains an expression that is used to construct the result of the FLWR expression. The `return` clause is invoked once for every tuple generated by the `for` and `let` clauses, after eliminating any tuples that do not satisfy the conditions of a `where` clause. The expression in the `return` clause is evaluated once for every invocation, and the result of the FLWR expression is an ordered sequence containing the results of these invocations.

Expressions in `for`, `let` and `return` clauses can be any top-level expression, with the restriction that: expressions in `for` and `let` clauses should not contain constructor expressions, at any level.

A variable name may not be used before it is bound, nor may it be used in the expression to which it is bound. Any variable bound in a `for` or `let` clause is in scope until the end of the FLWR expression in which it is bound. If the variable name used in the binding was already bound in the current scope, the variable name refers to the newly bound variable until that variable goes out of scope. At this point, the variable name again refers to the variable of the prior binding.

Although `for` and `let` both bind variables, the manner in which variables are bound is quite different. In a `let` clause, the variable is bound directly to the expression, and it is bound to the expression as a whole. Consider the following query, based on the data presented in the earlier Example section:

```
let $users := collection("USERS")/USERS/NAME
return <out>{$users}</out>
```

The variable `$users` is bound to the expression `collection("USERS")/USERS/NAME`, i.e. the sequence containing all the NAME subelements in the USERS table. There are no `for` clauses, so the `let` clause generates one tuple that contains the variable binding of $users. The `return` clause is invoked for this tuple, creating the following output:

```
<out>
    <NAME>Tom Jones</NAME>
    <NAME>Mary Doe</NAME>
    <NAME>Dee Linquent</NAME>
    <NAME>Roger Smith</NAME>
    <NAME>Jack Sprat</NAME>
    <NAME>Rip Van Winkle</NAME>
</out>
```

Now consider a similar query which contains a `for` clause instead of a `let` clause:

```
for $user in collection("USERS")/USERS/NAME
return <out>{$user}</out>
```

The variable `$user` is associated with the expression `collection("USERS")/USERS/NAME`, from which the variable bindings of $user will be drawn. When only one expression is present, the Cartesian product is equivalent to the sequence of values returned by that expression. In this example, the variable $user is bound six times, to each NAME subelement in the USERS table. One tuple is generated for each of these variable bindings, and the `return` clause is invoked for each tuple, creating the following output:

```
<out>
    <NAME>Tom Jones</NAME>
</out>
<out>
    <NAME>Mary Doe</NAME>
</out>
<out>
    <NAME>Dee Linquent</NAME>
</out>
<out>
    <NAME>Roger Smith</NAME>
</out>
<out>
    <NAME>Jack Sprat</NAME>
</out>
<out>
```

```
    <NAME>Rip Van Winkle</NAME>
</out>
```

Note that the above result is not a well-formed XML document, as it contains multiple root elements. It is necessary to enclose the result fragments into an enclosing element to produce a valid XML document.

A FLWR Expression may contain multiple `for` clauses. In this case, the tuples of variable bindings are drawn from the Cartesian product of the sequences returned by the expressions in all the `for` clauses. The ordering of the tuples is governed by the ordering of the sequences from which they were formed, working from left to right.

The following expression illustrates how tuples are generated from the Cartesian product of expressions in a `for` clause. For each user who has offered an item for auction, it returns the user name and the item description.

```
for $u in collection("USERS")/USERS,
    $i in collection("ITEMS")/ITEMS
where $u/USERID = $i/OFFERED_BY
return
  <result>
    { $u/NAME }
    { $i/DESCRIPTION }
  </result>
```

Here is the result of the above expression.

```
<result>
  <NAME>Tom Jones</NAME>
  <DESCRIPTION>Red Bicycle</DESCRIPTION>
</result>
<result>
  <NAME>Tom Jones</NAME>
  <DESCRIPTION>Tricycle</DESCRIPTION>
</result>
<result>
  <NAME>Tom Jones</NAME>
  <DESCRIPTION>Broken Bicycle</DESCRIPTION>
</result>
<result>
  <NAME>Mary Doe</NAME>
  <DESCRIPTION>Motorcycle</DESCRIPTION>
</result>
<result>
  <NAME>Mary Doe</NAME>
  <DESCRIPTION>Old Bicycle</DESCRIPTION>
</result>
<result>
  <NAME>Dee Linquent</NAME>
  <DESCRIPTION>Tennis Racket</DESCRIPTION>
</result>
<result>
  <NAME>Dee Linquent</NAME>
```

```
  <DESCRIPTION>Helicopter</DESCRIPTION>
</result>
<result>
  <NAME>Roger Smith</NAME>
  <DESCRIPTION>Racing Bicycle</DESCRIPTION>
</result>
```

The following expression is a slightly modified query, showing the restructuring capabilities of XQuery. For each user, it returns the user name and the description of all items it has offered for auction.

```
for $u in collection("USERS")/USERS
return
  <result>
    { $u/NAME }
    { for $i in collection("ITEMS")/ITEMS
      where $u/USERID = $i/OFFERED_BY
      return $i/DESCRIPTION }
  </result>
```

Here is the result of the above expression. Note that the result for users having not offered items does not contain descriptions, although the user name is still present. This is an example of how XQuery handles the relational concept of outer join.

```
<result>
  <NAME>Tom Jones</NAME>
  <DESCRIPTION>Red Bicycle</DESCRIPTION>
  <DESCRIPTION>Tricycle</DESCRIPTION>
  <DESCRIPTION>Broken Bicycle</DESCRIPTION>
</result>
<result>
  <NAME>Mary Doe</NAME>
  <DESCRIPTION>Motorcycle</DESCRIPTION>
  <DESCRIPTION>Old Bicycle</DESCRIPTION>
</result>
<result>
  <NAME>Dee Linquent</NAME>
  <DESCRIPTION>Tennis Racket</DESCRIPTION>
  <DESCRIPTION>Helicopter</DESCRIPTION>
</result>
<result>
  <NAME>Roger Smith</NAME>
  <DESCRIPTION>Racing Bicycle</DESCRIPTION>
</result>
<result>
  <NAME>Jack Sprat</NAME>
</result>
<result>
  <NAME>Rip Van Winkle</NAME>
</result>
```

> **Note on the use of views**: The above restructuration uses unique identifiers for rows to group results according to the external loop variables. When using views (or synonyms), XQuark Bridge cannot use such identifiers; in that case, it uses the complete row as identifier, and automatically removes duplicate rows.

Sorting Expressions

A sorting expression provides a way to control the order of items in a sequence.

```
SortExpr     ::= (PathExpr | FLWRExpr)
                 "sortby" "(" SortSpecList ")"
SortSpecList ::= (PathExpr | StepExpr) SortModifier
                 ("," SortSpecList)?
SortModifier ::= ("ascending" | "descending")?
```

The value of the expression on the left side of the `sortby` keyword is called the **input expression**. The items in the input expression are called **input items**. The result of the sorting expression is called the **output expression**. The output expression contains all the input items, retaining their original identities (if any), but possibly in a different order.

The expressions on the right side of the `sortby` keyword are called **ordering expressions**. For each input item, the ordering expressions are evaluated with an **inner focus** derived from the input item. The input items are then reordered according to the values of their respective ordering expressions. If more than one ordering expression is specified, the leftmost ordering expression controls the primary sort, followed by the remaining ordering expressions from left to right. Each ordering expression can be followed by the keyword `ascending` or `descending`, which specifies the direction of the sort (`ascending` is the default).

The process of evaluating and comparing the ordering expressions is based on the following rules:

- **Atomization** is applied to the result of each ordering expression, resulting in a single atomic value or an empty sequence for each operand ordering expression. Static type checking is enforced: if the compiler detects that an ordering expression may result in a sequence of length greater than 1, a type exception is thrown.
- If the result of an ordering expression has the type `xs:anySimpleType` (such as character data in a schemaless document), it is cast to the type `xs:string`.

- Each ordering expression must return values of the same type for all input items, and this type must be a (possibly optional) atomic type for which the > operator is defined--otherwise, the error value is returned.
- For the purpose of the following rule, an ordering value that is an empty sequence is treated as greater than any non-empty ordering value.
- If V1 and V2 are the values of an ordering expression for input items I1 and I2 respectively, then:
  - If the ordering expression is ascending, and if V2 > V1 is true, then I1 precedes I2 in the output sequence.
  - If the ordering expression is descending, and if V1 > V2 is true, then I1 precedes I2 in the output sequence.
  - If neither V1 > V2 nor V2 > V1 is true, then the order of I1 and I2 in the output sequence is implementation-defined.

Here are some examples of ordering expressions:

This example lists all bids in the auction database, ordered first by bid, then by user name.

```
for $b in collection("BIDS")/BIDS,
    $u in collection("USERS")/USERS,
    $i in collection("ITEMS")/ITEMS
where $b/USERID = $u/USERID
      and $b/ITEMNO=$i/ITEMNO
return
  <result>
    { $u/NAME }
    { $b/BID }
    { $i/DESCRIPTION }
  </result>
sortby (BID, NAME)
```

Ordering may be specified at multiple levels of a query result.

```
<result>
  { for $i in collection("ITEMS")/ITEMS
    return
      <item>
        <name> { data($i/DESCRIPTION)} </name>
        <reserve>
          { data($i/RESERVE_PRICE) }
        </reserve>
        <bids>
          { for $b in collection("BIDS")/BIDS,
                $u in collection("USERS")/USERS
            where $b/ITEMNO = $i/ITEMNO
                  and $b/USERID = $u/USERID
            return
              <bid>
                <name> { data($u/NAME) } </name>
                <value> { data($b/BID) } </value>
              </bid>
            sortby(value)
```

```
            }
          </bids>
        </item>
      sortby(reserve)
    }
</result>
```

Quantified Expressions

Quantified expressions support existential and universal quantification. The value of a quantified expression is always `true` or `false`.

```
QuantifiedExpr ::=   ("some" | "every")
                     Variable "in" Expr
                     ("," Variable "in" Expr)*
                     "satisfies" OrExpr
```

A **quantified expression** begins with a **quantifier**, which is the keyword `some` or `every`, followed by one or more in-clauses that are used to bind variables, followed by the keyword `satisfies` and a test expression. Each in-clause associates a variable with an expression that returns a sequence of values. Any top-level expression can be used, with the restriction that it should not contain constructor expressions.

As in the case of a for-clause in a FLWR-expression, the in-clauses generate tuples of variable bindings, using values drawn from the Cartesian product of the sequences returned by the binding expressions. Conceptually, the test expression is evaluated for each tuple of variable bindings. Results depend on the **effective boolean values** of the test expressions. The value of the quantified expression is defined by the following rules:

- If the quantifier is `some`, the quantified expression is `true` if at least one evaluation of the test expression has the **effective boolean value** `true`; otherwise the quantified expression is `false`. This rule implies that, if the in-clauses generate zero binding tuples, the value of the quantified expression is `false`.
- If the quantifier is `every`, the quantified expression is `true` if every evaluation of the test expression has the **effective boolean value** `true`; otherwise the quantified expression is `false`. This rule implies that, if the in-clauses generate zero binding tuples, the value of the quantified expression is `true`.

The order in which test expressions are evaluated for the various binding tuples is implementation-defined. If the quantifier is `some`, an implementation may return `true` as soon as it finds one binding tuple for which the test expression has an effective Boolean value of `true`, and it may return an error as soon as it finds one binding tuple for which the test expression returns an error. Similarly, if the quantifier is `every`, an implementation may return `false` as soon as it finds one binding tuple for

which the test expression has an effective Boolean value of `false`, and it may return an error as soon as it finds one binding tuple for which the test expression returns an error. As a result of these rules, the value of a quantified expression is not deterministic in the presence of errors.

Here are some examples of quantified expressions:

This expression returns the users, if any, that have bidded on every item:

```
<frequent_bidder>
{
  for $u in collection("USERS")/USERS
  where
    every $item in collection("ITEMS")/ITEMS
satisfies
      some $b in collection("BIDS")/BIDS
satisfies
        $item/ITEMNO = $b/ITEMNO
         and $u/USERID = $b/USERID
  return $u/NAME
}
</frequent_bidder>
```

This expression returns the users, if any, that have bidded on at least one item:

```
<bidder>
{
  for $u in collection("USERS")/USERS
  where
    some $item in collection("ITEMS")/ITEMS
satisfies
      some $b in collection("BIDS")/BIDS
satisfies
        $item/ITEMNO = $b/ITEMNO
        and $u/USERID = $b/USERID)
  return $u/NAME
}
</bidder>
```

Part 6

Built-in XQuery Functions

[XML Schema Part 2] defines a number of primitive and derived datatypes, collectively known as built-in datatypes. This section defines operations on those datatypes for use in XQuery. It also discusses operations on nodes and node sequences as defined in the [XQuery 1.0 and XPath 2.0 Data Model] for use in XQuery.

> **Note:** XQuark Bridge only supports a subset of the built-in user functions defined by the XQuery draft standard. The complete set is described in [XQuery 1.0 and XPath 2.0 Functions and Operators].

Accessors

The [XQuery 1.0 and XPath 2.0 Data Model] describes accessors on different types of nodes and defines their semantics. In XQuark Bridge, two of these accessors are exposed to the user through the functions described below.

Function	Accessor	Accepts	Returns
`string`	`string-value`	a sequence, a node of any kind, or a simple value	string
`data`	`typed-value`	any kind of node	a typed sequence of atomic values

string

```
string(item* $srcval) => string
```

Returns the value of `$srcval` represented as a string.

If `$srcval` is the empty sequence, the empty string is returned.

If `$srcval` is a node, the return value depends on the node type:

- for an attribute node or an element node with simple content, the string value of the node's sequence of atomic values is returned, as described below.
- for a element node with complex content, a type exception occurs, resulting in the error value.

> **Compatibility note:** The XQuery draft standard specifies that the string value of a complex element node is the concatenation of the string values of its descendant text nodes in document order. This is not supported by XQuark Bridge.

If `$srcval` is an atomic value, the function returns the canonical lexical representation of the typed value, as defined in [XML Schema Part 2], except in the case listed below:

- If the type of `$srcval` is `xs:decimal`, and the value is equal to an integer, then the function returns the canonical representation of that integer. This special rule allows integers to be displayed without the decimal point.

If `$srcval` is a sequence of more than one item, a type exception occurs, resulting in the error value.

data

```
data(node* $srcval) => value*
```

Returns the typed-value of each node in `$srcval`. Each node in `$srcval` is processed as follows.

The static type of the result for each node is determined by the static type of the value that is extracted.

If `$srcval` is not a element, attribute or text node, returns the error value.

If `$srcval` is a text node, returns the string content of the text node with type annotation `xs:anySimpleType`.

If `$srcval` is an attribute node defined to have `xs:anySimpleType`, or an element node with simple content defined to have `xs:anySimpleType`, returns its string value with type annotation `xs:anySimpleType`.

If `$srcval` is an element or attribute node with a simple type other than `xs:anySimpleType` or with a complex type with simple content other than `xs:anySimpleType`, returns the node's typed value which is a sequence of atomic values.

If `$srcval` is an element node with complex content, returns the error value.

Constructors and Functions on Numbers

This section discusses arithmetic operators on the numeric datatypes defined in [XML Schema Part 2].

Numeric Types

The operators described in this section are defined on the following numeric types:

- `xs:decimal`
- `xs:integer`
- `xs:double`

They also apply to types derived by restriction from these types.

Numeric Constructors

The following constructors are defined on the above numeric types. Each constructor takes a single `xs:string` literal as argument. Leading and trailing whitespace, if present, is stripped from the literal before the value is constructed.

Constructor	Meaning
`decimal`	Produces a decimal value by parsing and interpreting a string.
`integer`	Produces an integer value by parsing and interpreting a string.
`double`	Produces a double value by parsing and interpreting a string.

If the argument string passed to a constructor results in an error (for example, if it contains a letter other than "E" or "e"), the constructor returns the error value.

decimal

```
decimal(string $srcval) => decimal
```

Returns the `decimal` value that is represented by the characters contained in the value of `$srcval`. For this constructor, `$srcval` must be a string literal.

If the value of `$srcval` is not a valid lexical representation for the `decimal` type as specified in [XML Schema Part 2], then the error value is returned.

If the number of characters contained in the value of `$srcval` that are digits is greater than the maximum number of decimal digits supported by the implementation, then the error value is returned.

Examples:

- `decimal('123.5')` returns the `decimal` value corresponding to one hundred twenty three and one-half.
- `decimal('12.5E2')` returns the error value, since the use of the letter "E" is prohibited in the constructor for the `decimal` type.
- `decimal(' 12.5 ')` returns the `decimal` value corresponding to twelve and one-half.

integer

```
integer(xs:string $srcval) => integer
```

Returns the `integer` value that is represented by the characters contained in the value of `$srcval`. For this constructor, `$srcval` must be a string literal.

If the value of `$srcval` is not a valid lexical representation for the `integer` type as specified in [XML Schema Part 2], then the error value is returned.

If the number of characters contained in the value of `$srcval` that are digits is greater than the maximum number of digits supported by the implementation, then the error value is returned.

Examples:

- `integer('-123')` returns the `integer` value corresponding to negative one hundred twenty three.
- `integer('123.5')` returns the error value, since the use of a decimal point is prohibited in the constructor for the `integer` type.

double

```
double(xs:string $srcval) => double
```

Returns the `double` value that is represented by the characters contained in the value of `$srcval`. For this constructor, `$srcval` must be a string literal.

If the value of `$srcval` is not a valid lexical representation for the `double` type as specified in [XML Schema Part 2], then the error value is returned. Note that XQuark Bridge does not support the special values `NaN`, `INF`, `+INF` and `-INF`.

Examples:

- `double('510E2')` returns the `double` value corresponding to fifty one thousand.
- `double('15.25')` returns the `double` value corresponding to fifteen and a quarter.
- `double('51D1')` returns the error value, since the use of the letter "D" is prohibited in the constructor for the `double` type.

Functions on Numeric Values

The following functions are defined on numeric types. Each function returns an integer except:

- If the argument is the empty sequence, the empty sequence is returned.
- The `abs()` function returns an `xs:double`.

floor

```
floor(xs:double? $srcval) => xs:integer?
```

Returns the largest (closest to positive infinity) integer that is not greater than the value of `$srcval`.

If the argument is the empty sequence, returns the empty sequence.

Examples:

- `floor(10.5)` returns 10.
- `floor(-10.5)` returns -11.

ceiling

```
ceiling(xs:double? $srcval) => xs:integer?
```

Returns the smallest (closest to negative infinity) number that is not smaller than the value of `$srcval`and that is an integer.

If the argument is the empty sequnce, returns the empty sequence.

Examples:

- `ceiling(10.5)` returns 11.
- `ceiling(-10.5)` returns -10.

round

```
round(xs:double? $srcval) => xs:integer?
```

Returns the number that is closest to the argument and that is an integer. More formally, `round(x)` produces the same result as `floor(x+0.5)`. If there are two such numbers, then the one that is closest to positive infinity is returned.

If the argument is the empty sequence, returns the empty sequence.

Examples:

- `round(2.5)` returns 3.
- `round(2.4999)` returns 2.
- `round(-2.5)` returns -2.

abs

```
abs(xs:double? $srcval) => xs:double?
```

Returns the absolute value of the argument.

If the argument is the empty sequence, returns the empty sequence.

Examples:

- `abs(2.5)` returns 2.5.
- `abs(-3)` returns 3.

Constructors and Functions on Strings

This section discusses operators on the [XML Schema Part 2] `xs:string` datatype. They also apply to types derived by restriction from this type.

String Constructor

The following constructor is defined on the `xs:string` type. This constructor takes a single `string` literal as argument.

Constructor	Meaning
`string`	Produces a string value by parsing and interpreting a supplied string.

string

```
string(xs:string $srcval) => xs:string
```

Returns a `string` value that is the value of `$srcval`. The more general accessor-based function `string` returns the string value for several kinds of

input arguments. If the input argument is a string it just returns the argument string. Thus, this constructor can be correctly perceived as a "no-op", but is included for the sake of orthogonality.

Functions on String Values

The following functions are defined on these string types. Several of these functions use a default collation, which is the collation used by the underlying relational database.

Function	Meaning
`concat`	Concatenates two or more character strings.
`starts-with`	Indicates whether the value of one string begins with the characters of the value of another string.
`ends-with`	Indicates whether the value of one string ends with the characters of the value of another string.
`contains`	Indicates whether the value of one string contains the characters of the value of another string.
`substring`	Returns a string located at a specified place in the value of a string.
`string-length`	Returns the length of the argument.
`upper-case`	Returns the upper-cased value of the argument.
`lower-case`	Returns the lower-cased value of the argument.

concat

```
concat() => xs:string

concat(xs:string? $op1) => xs:string

concat(xs:string? $op1, xs:string? $op2, ...)
=> xs:string
```

Accepts zero or more strings as arguments. Returns the string that is the concatenation of the values of its arguments. The resulting string might not be normalized in any Unicode or W3C normalization. If called with no arguments, returns the zero-length string. If any of the arguments is the empty sequence it is treated as the zero-length string.

Examples:

- `concat('abc', 'def')` returns "`abcdef`".
- `concat('abc')` returns `abc`.

- `concat('abc', 'def', 'ghi', 'jkl', 'mno')` returns `"abcdefghijklmno"`.

starts-with

```
starts-with(xs:string? $op1, xs:string? $op2)
=> xs:boolean?
```

Returns a boolean indicating whether or not the value of `$op1` starts with a string that is equal to the value of `$op2`.

If the value of `$op2` is the zero-length string, then the function returns `true`. If the value of `$op1` is the zero-length string and the value of `$op2` is not the zero-length string, then the function returns `false`.

If the value of `$op1` or `$op2` is the empty sequence, the empty sequence is returned.

Examples:

- `starts-with("goldenrod", "gold")` returns `true`.
- `starts-with("goldenrod", "")` returns `true`.
- `starts-with("goldenrod", "rod")` returns `false`.

ends-with

```
ends-with(xs:string? $op1, xs:string? $op2)
=> xs:boolean?
```

Returns a boolean indicating whether or not the value of `$op1` ends with a string that is equal to the value of `$op2`.

If the value of `$op2` is the zero-length string, then the function returns `true`. If the value of `$op1` is the zero-length string and the value of `$op2` is not the zero-length string, then the function returns `false`.

If the value of `$op1` or `$op2` is the empty sequence, the empty sequence is returned.

Examples:

- `ends-with("goldenrod","rod")` returns `true`.
- `ends-with("", "rod")` returns `false`.

contains

```
contains(xs:string? $op1, xs:string? $op2)
=> xs:boolean?
```

Returns a boolean indicating whether or not the value of `$op1` contains (at the beginning, at the end, or anywhere within) a string equal to the value of `$op2`.

If the value of `$op2` is the zero-length string, then the function returns `true`. If the value of `$op1` is the zero-length string and the value of `$op2` is not the zero-length string, then the function returns `false`.

If the value of `$op1` or `$op2` is the empty sequence, the empty sequence is returned.

substring

```
substring(xs:string? $sourceString,
          xs:decimal? $startingLoc)
=> xs:string?

substring(xs:string? $sourceString,
          xs:decimal? $startingLoc,
          xs:decimal? $length)
=> xs:string?
```

Returns the portion of the value of `$sourceString` beginning at the position indicated by the value of `$startingLoc` and continuing for the number of characters indicated by the value of `$length`. More specifically, returns the characters in `$sourceString` whose position `$p` obeys:

```
round($startingLoc) <= $p < round($startingLoc + $length)
```

If `$length` is not specified, the substring identifies characters to the end of `$sourceString`.

If `$length` is greater than the number of characters in the value of `$sourceString` following `$startingLoc`, the substring identifies characters to the end of `$sourceString`.

The first character of a string is located at position 1 (not position 0).

If the value of `$startingLoc` is negative or greater than the length of `$sourceString`, the behavior is undefined.

If the value of any of the three parameters is the empty sequence, the empty sequence is returned.

Examples:

- `substring("motor car", 6)` returns `" car"`.
- `substring("metadata", 4, 3)` returns `"ada"`.

string-length

```
string-length(xs:string? $srcval) => xs:integer?
```

Returns an integer equal to the length in characters of the value of `$srcval`. If the value of `$srcval` is the empty sequence, the empty sequence is returned.

Examples:

- `string-length("motor car")` returns `9`.

upper-case

```
upper-case(xs:string? $srcval) => xs:string?
```

Returns the value of `$srcval` after translating every lower-case letter to its upper-case correspondent. Every lower-case letter that does not have an upper-case correspondent, and every character that is not a lower-case letter, is included in the returned value in its original form.

If the value of `$srcval` is the empty sequence, returns the empty sequence.

Examples:

- `upper-case("abCd0")` returns `"ABCD0"`.

lower-case

```
lower-case(xs:string? $srcval) => xs:string?
```

Returns the value of `$srcval` after translating every upper-case letter to its lower-case correspondent. Every upper-case letter that does not have a lower-case correspondent, and every character that is not an upper-case letter, is included in the output in its original form.

If the value of `$srcval` is the empty sequence, returns the empty sequence.

Examples:

- `lower-case("ABc!D")` returns `"abc!d"`.

Constructors and Functions on Booleans

This section discusses operators on the [XML Schema Part 2] `xs:boolean` datatype.

Boolean Constructors

The following constructors are defined on the boolean type.

Constructor	Meaning
`true`	boolean true value
`false`	boolean false value

true

```
true() => xs:boolean
```

Returns the boolean value `true`.

false

```
false() => xs:boolean
```

Returns the boolean value `false`.

Functions on Boolean Values

The following function is defined on boolean values:

Function	Meaning
`xf:not`	Inverts the boolean value of the argument. A () argument returns `true`.

not

```
not(item* $srcval) => xs:boolean
```

`$srcval` is first reduced to an effective boolean value by applying the following rules:

- If `$srcval` is the empty sequence, its effective boolean value is `false`.
- If `$srcval` is a single boolean value, it serves as its own effective boolean value.
- If `$srcval` is a sequence that contains at least one node, its effective boolean value is `true`.
- In any other case, a type exception is invoked.

Returns `true` if the effective boolean value is `false`, and `false` if the effective boolean value is `true`.

Examples:

- `not(true())` returns `false`.

Constructors and Functions on Dates and Times

This section discusses operations on the [XML Schema Part 2] date and time types.

Date and Time Types

The operators described in this section are defined on the following date and time types:

- `xs:dateTime`
- `xs:date`
- `xs:time`

Date and Time Constructors

The following constructors are defined on date and time datatypes. Each constructor takes a single `string` literal as argument. Leading and trailing whitespace, if present, is stripped from the literal before the value is constructed.

Constructor	Meaning
`dateTime`	Returns a dateTime type derived by parsing and interpreting a string value.
`date`	Returns a date type derived by parsing and interpreting a string value.
`time`	Returns a time type derived by parsing and interpreting a string value.

dateTime

```
dateTime(xs:string $srcval) => xs:dateTime
```

If the value of `$srcval` conforms to the lexical representation of a `xs:dateTime` as defined in [XML Schema Part 2], the constructor returns the `dateTime` corresponding to that representation. Otherwise, the constructor returns the error value.

Examples:

- `dateTime("1999-05-31T05:00:00")` returns a `xs:dateTime` value corresponding to the 31st. of May, 1999 at 5:00 am in an unspecified timezone.
- `dateTime("1999-05-31T13:20:00-05:00")` returns a `xs:dateTime` value corresponding to 1:20 pm on May the 31st, 1999 for a timezone which is 5 hours behind Coordinated Universal Time (UTC).

date

```
date(xs:string $srcval) => xs:date
```

If the value of `$srcval` conforms to the lexical representation of a `xs:date` as defined in [XML Schema Part 2], the constructor returns the `date` corresponding to that representation. Otherwise, the constructor returns the error value.

Examples:

- `date("2001-05-31")` returns a `xs:date` value corresponding to the 31st of May, 2001.
- `date("2001-04-31")` returns an error.

time

```
time(xs:string $srcval) => xs:time
```

If the value of `$srcval` conforms to the lexical representation of a `xs:time` as defined in [XML Schema Part 2], the constructor returns the `time` corresponding to that representation. Otherwise, the constructor returns the error value.

Examples:

- `time("11:33:24")` returns a `xs:time` value corresponding to 33 minutes and 24 seconds past 11 o'clock in an unspecified timezone.
- `time("23:33:24.35-05:00")` returns a `xs:time` value corresponding to 33 minutes and 24.35 seconds past 23 o'clock for a timezone which is 5 hours behind Coordinated Universal Time (UTC).

Functions on Nodes

This section discusses operators on `nodes`. `Nodes` have been introduced in The standard XQuery data model and are formally defined in [XQuery 1.0 and XPath 2.0 Data Model].

Functions on Nodes

The following function is defined on nodes:

Function	Meaning
number	Returns the value of the context node or the specified node converted to a number.

number

```
number(node? $srcval) => xs:double?
```

Returns the value of the node indicated by `$srcval` converted to a `xs:double`. If the value of the node is not a valid lexical representation of a numeric simple type as defined in [XML Schema Part 2], then the function returns the error value.

If the value of `$srcval` is the empty sequence, the empty sequence is returned.

Functions on Sequences

This section discusses operators on `sequences`, i.e. ordered collections of zero or more `items`. The terms `sequence` and `item` have been introduced in The standard XQuery data model and are defined formally in [XQuery 1.0 and XPath 2.0 Data Model].

Functions on Sequences

The following functions are defined on sequences.

Function	Meaning
empty	Indicates whether or not the provided sequence is empty.
exists	Indicates whether or not the provided sequence is not empty.
distinct-values	Returns a sequence in which all redundant duplicate elements, based on value equality, have been deleted. The specific node in a collection of redundant duplicate nodes that is retained in implementation-dependent.

empty

```
empty(item* $srcval) => xs:boolean
```

If the the value of `$srcval` is the empty sequence, the function returns `true`; otherwise, the function returns `false`.

exists

```
exists(item* $srcval) => xs:boolean
```

If the the value of `$srcval` is not the empty sequence, the function returns `true`; otherwise, the function returns `false`.

distinct-values

```
distinct-values(item* $srcval) => item*
```

`$srcval` must contain either simple values or nodes, not both. If the sequence contains both simple values and nodes, then the function returns the error value.

If `$srcval` contains only nodes, returns the sequence that results from removing from `$srcval` all but one of a set of nodes that are equal to one other, based on the following comparison function. Two nodes are considered equal if:

- they have the same node type (i.e. element or attribute),
- they have the same name,
- they have the same value, as obtained by the `data()` function. Note that the use of the data() function implies that it is an error to apply the `distinct-values` function to sequences containing element nodes with complex content.

The specific node in a collection of nodes having equal values that is retained is implementation-dependent.

If `$srcval` contains only values, returns the sequence that results from removing from `$srcval` all but one of a set of values that are equal to one other.

If `$srcval` is the empty sequence, returns the empty sequence.

Aggregate Functions

Aggregate functions take a sequence as argument and return a single value computed from values in the sequence. Except for count, if the sequence contains nodes, the value is extracted from the node and used in the computation.

Function	Meaning

Function	Meaning
count	Returns the number of items in the sequence.
avg	Returns the average of a sequence of numbers.
max	Returns the object with maximum value from a collection of comparable objects.
min	Returns the object with minimum value from a collection of comparable objects.
sum	Returns the sum of a sequence of numbers.

count

```
count(item* $srcval) => xs:unsignedInt
```

Returns the number of items in the value of `$srcval`. Returns 0 if `$srcval` is the empty sequence.

avg

```
avg(item* $srcval) => xs:double?
```

If `$srcval` contains nodes, the value of each node is extracted using the `data()` function. Values that equal the empty sequence are discarded. If after this, `$srcval` contains only numbers, `avg()` returns the average of the numbers (computed as `sum($srcval) div count($srcval)`). If `$srcval` is the empty sequence, the empty sequence is returned.

If, after extracting the values from nodes, `$srcval` does not contain only numbers, the function returns the error value.

max

```
max(item* $srcval) => xs:anySimpleType?
```

If `$srcval` contains nodes, the value of each node is extracted using the `data()` function. Values that equal the empty sequence are discarded. If, after this, `$srcval` is the empty sequence, the empty sequence is returned. After extracting the values from nodes, `$srcval` must contain only values of a single type (for numeric values, the type promotion rules can be used to promote them to a single type). Otherwise, the function returns the error value.

max returns the item in the value of `$srcval` whose value is greater than the value of every other item in the value of `$srcval`. If there are two or more such items, then the specific item whose value is returned is implementation-dependent.

min

```
min(item* $srcval) => xs:anySimpleType?
```

If `$srcval` contains nodes, the value of each node is extracted using the the `data()` function. Values that equal the empty sequence are discarded. If, after this, `$srcval` is the empty sequence, the empty sequence is returned. After extracting the values from nodes, `$srcval` must contain only values of a single type (for numeric values, the type promotion rules can be used to promote them to a single type) Otherwise, the function returns the error value.

min returns the item in the value of `$srcval` whose value is less than the value of every other item in the value of `$srcval`. If there are two or more such items, then the specific item whose value is returned is implementation-dependent.

sum

```
sum(item* $srcval) => xs:double
```

If `$srcval` contains nodes, the value of each node is extracted using the `data()` function. Values that equal the empty sequence are discarded. If, after this, `$srcval` contains only numbers, xf:sum() returns the sum of the numbers. If it is the empty sequence, 0.0 is returned.

If, after extracting the values from nodes, `$srcval` does not contain only numbers, the function returns the error value.

Functions that Generate Sequences

Function	Meaning
`collection`	Returns the collection (a sequence of document nodes) retrieved using the string specified as its argument.
`input`	Returns the input sequence.

collection

```
collection(xs:string $srcval) => node*
```

Takes a string as argument and returns the sequence of document nodes generated from the corresponding relational table. `$srcval` should be the name of an existing relational table (after applying to it the transformations specified in the XQuark Bridge configuration file), prefixed by the relational schema name, if several schemas are in use. The function returns the error value if `$srcval` does not resolve to a valid table name.

Note that `collection()` cannot be used by itself and must necessarily be followed by a navigation expression (StepExpr or PathExpr). This expression should always start with a step whose local name is equal to the name of the table.

Examples:

- `collection("ITEMS")/ITEMS` returns the content of the ITEMS table, as a sequence of `ITEMS` element nodes.
- `collection("AUCTION.ITEMS")/ITEMS` returns the content of the ITEMS table located in the AUCTION relational schema, as a sequence of `ITEMS` element nodes.

The above example assumes that the XML schema generated from the AUCTION relational schema has no target namespace. Otherwise, the query should be written:

```
namespace ns = "http://myauctionschema"
collection("AUCTION.ITEMS")/ns:ITEMS
```

input

```
input() => node*
```

Returns the input sequence, i.e. the union of all collections (i.e. wrapped tables) available in the underlying relational database.

Note that `input()` cannot be used by itself and must necessarily be followed by a navigation expression (StepExpr or PathExpr). This expression selects the table to be accessed based on the name specified in the first navigation step.

In XQuark Bridge, the `input()` function is implicit in outermost navigation expressions.

Examples:

- `input()/ITEMS` returns the content of the ITEMS table, as a sequence of `ITEMS` element nodes.
- `namespace ns = "http://myauctionschema"`
  `input()/ns:ITEMS`
  returns the content of theAUCTION.ITEMS table, as a sequence of `ns:ITEMS` element nodes, assuming that the target namespace associated to the AUCTION relation schema is "http://myauctionschema".

`/ITEMS` and `/ns:ITEMS` are respectively equivalent to the two previous expressions.

Context Functions

The following function is defined to obtain information from the evaluation context.

Function	Meaning
`current-dateTime`	Returns the current dateTime.

current-dateTime

```
current-dateTime() => xs:dateTime
```

Returns the `xs:dateTime` that is current at some time during the evaluation of the XQuery expression in which `current-dateTime()` is executed. All invocations of `current-dateTime()` that are executed during the course of a single outermost XQuery expression return the same value. The precise instant during that XQuery expression's evaluation represented by the value of `current-dateTime()` is not defined.

Index

Appendix A – XML Schema for the XQuark Bridge configuration file

```
<?xml version="1.0"?>
<schema
  xmlns="http://www.w3.org/2001/XMLSchema"
  xmlns:ds="http://www.xquark.org/Bridge/1.0/Datasource"
  targetNamespace=
    " http://www.xquark.org/Bridge/1.0/Datasource ">

  <simpleType name="caseType">
    <restriction base="string">
      <enumeration value="mixed"/>
      <enumeration value="lower"/>
      <enumeration value="upper"/>
    </restriction>
  </simpleType>

  <simpleType name="elementFormType">
    <restriction base="string">
      <enumeration value="qualified"/>
      <enumeration value="unqualified"/>
    </restriction>
  </simpleType>

  <simpleType name="charType">
    <restriction base="string">
      <length value="1"/>
    </restriction>
  </simpleType>

  <simpleType name="substType">
    <restriction base="string">
      <pattern value="[\c-[:]]*" />
    </restriction>
  </simpleType>

  <complexType name="relationalStructType">
    <attribute name="regex" type="string" />
    <attribute name="name" type="string" />
  </complexType>

  <complexType name="aliasedRelationalStructType">
    <complexContent>
      <extension base="ds:relationalStructType">
        <attribute name="alias" type="NCName" />
      </extension>
    </complexContent>
  </complexType>

  <complexType name="excludedTableType">
    <complexContent>
      <extension base="ds:relationalStructType"/>
```

```
    </complexContent>
  </complexType>

  <complexType name="excludedColumnType">
    <complexContent>
      <extension base="ds:relationalStructType"/>
    </complexContent>
  </complexType>

  <complexType name="includedColumnType">
    <complexContent>
      <extension base="ds:aliasedRelationalStructType"/>
    </complexContent>
  </complexType>

  <complexType name="includedTableType">
    <complexContent>
      <extension base="ds:aliasedRelationalStructType">
        <sequence>
          <element name="includes" minOccurs="0">
            <complexType>
              <sequence>
                <element name="column" maxOccurs="unbounded"
                         type="ds:includedColumnType" />
              </sequence>
            </complexType>
          </element>
          <element name="excludes" minOccurs="0">
            <complexType>
              <sequence>
                <element name="column" maxOccurs="unbounded"
                         type="ds:excludedColumnType" />
              </sequence>
            </complexType>
          </element>
        </sequence>
      </extension>
    </complexContent>
  </complexType>

  <element name="datasource">
    <complexType>
      <sequence>
        <element name="description" minOccurs="0" type="string" />
        <element name="url" type="string" />
        <element name="user" type="string" />
        <element name="password" type="string"/>
        <element name="substitutions" minOccurs="0">
          <complexType>
            <sequence>
              <element name="nameCase" minOccurs="0"
                       type="ds:caseType" />
              <element name="character" minOccurs="0"
                       maxOccurs="unbounded">
                <complexType>
                  <attribute name="value" type="ds:charType" />
                  <attribute name="subst" type="ds:substType" />
                </complexType>
```

```
              </element>
            </sequence>
          </complexType>
        </element>
        <element name="catalog" maxOccurs="unbounded">
          <complexType>
            <sequence>
              <element name="schema" maxOccurs="unbounded">
                <complexType>
                  <sequence>
                    <element name="includes" minOccurs="0">
                      <complexType>
                        <sequence>
                          <element name="table"
                                   maxOccurs="unbounded"
                                   type="ds:includedTableType" />
                        </sequence>
                      </complexType>
                    </element>
                    <element name="excludes" minOccurs="0">
                      <complexType>
                        <sequence>
                          <element name="table"
                                   maxOccurs="unbounded"
                                   type="ds:excludedTableType" />
                        </sequence>
                      </complexType>
                    </element>
                  </sequence>
                  <attribute name="name" type="string"
                             use="optional" />
                  <attribute name="targetNamespace" type="anyURI"
                             use="optional" />
                  <attribute name="elementFormDefault"
                             type="ds:elementFormType"
                             use="optional" />
                </complexType>
              </element>
            </sequence>
            <attribute name="name" type="string" use="optional" />
          </complexType>
        </element>
      </sequence>
      <attribute name="name" type="string"/>
    </complexType>
  </element>
</schema>
```

Appendix B – Complete BNF Grammar

```
Query                   ::= QueryProlog Expr
QueryProlog             ::= (NamespaceDecl
                            | DefaultNamespaceDecl)*
NamespaceDecl           ::= "namespace" NCName "=" StringLiteral
DefaultNamespaceDecl    ::= "default element namespace ="
                            StringLiteral
Expr                    ::= PrimaryExpr
                            | CommentExpr
                            | PathExpr
                            | AdditiveExpr
                            | Constructor
                            | FLWRExpr
                            | SortExpr
PrimaryExpr             ::= Literal
                            | FunctionCall
                            | Variable
                            | ParenthesizedExpr
Literal                 ::= NumericLiteral | StringLiteral
NumericLiteral          ::= IntegerLiteral
                            | DecimalLiteral
                            | DoubleLiteral
IntegerLiteral          ::= [0-9]+
DecimalLiteral          ::= ("." [0-9]+) | ([0-9]+ "." [0-9]*)
DoubleLiteral           ::= (("." [0-9]+)
                             | ([0-9]+ ("." [0-9]*)?))
                            ([e] | [E]) ([+] | [-])? [0-9]+
StringLiteral           ::= (["] ([^"])* ["])
                            | (['] ([^'])* ['])
Variable                ::= "$" NCName
ParenthesizedExpr       ::= "(" Expr ")"
FunctionCall            ::= QName "(" (Expr ("," Expr)*)? ")"
ExprComment             ::= "{--" [^}]* "--}"
PathExpr                ::= (PathExprRoot | StepExpr | PathExpr)?
                            "/" StepExpr
PathExprRoot            ::= InputExpr | Variable
InputExpr               ::= "collection("  StringLiteral  ")"
                            | "input()"
StepExpr                ::= Step Predicates
Step                    ::= (Axis NodeTest) | AbbreviatedStep
Axis                    ::= "child" "::"
                            | "attribute" "::"
                            | "self" "::"
NodeTest                ::= KindTest | NameTest
```

```
NameTest                ::= QName
KindTest                ::= "text" "(" ")"
                            "node" "(" ")"
Predicates              ::= ("[" OrExpr "]")*
AbbreviatedStep         ::= "." | (@ NameTest) | NodeTest
AdditiveExpr            ::= (AdditiveExpr ("+"  |  "-"))?
                            MultiplicativeExpr
MultiplicativeExpr      ::= (MultiplicativeExpr
                             ("*" | "div" | "mod"))?
                            UnaryExpr
UnaryExpr               ::= ("-" | "+")?
                            (PrimaryExpr
                             | PathExpr
                             | StepExpr)
CompExpr                ::= AdditiveExpr
                            ("=" | "!=" | "<" S | "<=" | ">" | ">=")
                            AdditiveExpr
OrExpr                  ::= (OrExpr "or")? AndExpr
AndExpr                 ::= (AndExpr "and")? BoolExpr
BoolExpr                ::= CompExpr
                            | PrimaryExpr
                            | PathExpr
                            | StepExpr
                            | QuantifiedExpr
Constructor             ::= ElementConstructor
                            | XmlComment
                            | XmlProcessingInstruction
                            | CdataSection
ElementConstructor      ::= "<" QName AttributeList
                            ("/>" | (">" ElementContent*
                            "</" QName ">"))
ElementContent          ::= Char
                            | "{{"
                            | "}}"
                            | ElementConstructor
                            | EnclosedExpr
                            | CdataSection
                            | CharRef
                            | PredefinedEntityRef
                            | XmlComment
                            | XmlProcessingInstruction
AttributeList           ::= (QName "=" AttributeValue)*
AttributeValue          ::= (["]
                             ("'" | AttrValueContent)*
                             ["])
                            |(['])
                              (""" | AttrValueContent)*
                              ['])
AttrValueContent        ::= Char
                            | CharRef
                            | "{{"
```

```
                              | "}}"
                              | EnclosedExpr
                              | PredefinedEntityRef
EnclosedExpr              ::= "{" Expr "}"
Char                      ::= [#x0009] | [#x000D]
                              | [#x000A] | [#x0020-#xFFFD])
CharRef                   ::= "&#" ([0-9]+
                              | ("x"([0-9] | [a-f]
                                 | [A-F])+)) ";"
PredefinedEntityRef       ::= "&" ("lt" | "gt" | "amp"
                              | "quot" | "apos") ";"
CdataSection              ::= "<![CDATA[" Char* "]]>"
XmlProcessingInstruction ::= "<?" NCName Char* "?>"
XmlComment                ::= "<!--" Char* "-->"
FLWRExpr                  ::= (ForClause | LetClause)+
                              WhereClause? "return" Expr
ForClause                 ::= "for" Variable "in" Expr
                              ("," Variable "in" Expr)*
LetClause                 ::= "let" Variable ":=" Expr
                              ("," Variable ":=" Expr)*
WhereClause               ::= "where" OrExpr
SortExpr                  ::= (PathExpr | FLWRExpr)
                              "sortby" "(" SortSpecList ")"
SortSpecList              ::= (PathExpr | StepExpr) SortModifier
                              ("," SortSpecList)?
SortModifier              ::= ("ascending" | "descending")?
QuantifiedExpr            ::= ("some" | "every")
                              Variable "in" Expr
                              ("," Variable "in" Expr)*
                              "satisfies" OrExpr
NCName                    ::= (Letter | '_') (NCNameChar)*
QName                     ::= (NCName ":")? NCName
NCNameChar                ::= Letter | Digit | '.' | '-' | '_'
                              | CombiningChar | Extender
Letter                    ::= BaseChar | Ideographic
BaseChar                  ::= [#x0041-#x005A] | [#x0061-#x007A]
                              | [#x00C0-#x00D6] | [#x00D8-#x00F6]
                              | [#x00F8-#x00FF] | [#x0100-#x0131]
                              | [#x0134-#x013E] | [#x0141-#x0148]
                              | [#x014A-#x017E] | [#x0180-#x01C3]
                              | [#x01CD-#x01F0] | [#x01F4-#x01F5]
                              | [#x01FA-#x0217] | [#x0250-#x02A8]
                              | [#x02BB-#x02C1] | #x0386 | [#x0388-
                              #x038A] | #x038C | [#x038E-#x03A1]
                              | [#x03A3-#x03CE] | [#x03D0-#x03D6]
                              | #x03DA | #x03DC | #x03DE | #x03E0
                              | [#x03E2-#x03F3] | [#x0401-#x040C]
                              | [#x040E-#x044F] | [#x0451-#x045C]
                              | [#x045E-#x0481] | [#x0490-#x04C4]
                              | [#x04C7-#x04C8] | [#x04CB-#x04CC]
```

```
                        | [#x04D0-#x04EB] | [#x04EE-#x04F5]
                        | [#x04F8-#x04F9] | [#x0531-#x0556]
                        | #x0559 | [#x0561-#x0586] | [#x05D0-
                        #x05EA] | [#x05F0-#x05F2] | [#x0621-
                        #x063A] | [#x0641-#x064A] | [#x0671-
                        #x06B7] | [#x06BA-#x06BE] | [#x06C0-
                        #x06CE] | [#x06D0-#x06D3] | #x06D5
                        | [#x06E5-#x06E6] | [#x0905-#x0939]
                        | #x093D | [#x0958-#x0961] | [#x0985-
                        #x098C] | [#x098F-#x0990] | [#x0993-
                        #x09A8] | [#x09AA-#x09B0] | #x09B2
                        | [#x09B6-#x09B9] | [#x09DC-#x09DD]
                        | [#x09DF-#x09E1] | [#x09F0-#x09F1]
                        | [#x0A05-#x0A0A] | [#x0A0F-#x0A10]
                        | [#x0A13-#x0A28] | [#x0A2A-#x0A30]
                        | [#x0A32-#x0A33] | [#x0A35-#x0A36]
                        | [#x0A38-#x0A39] | [#x0A59-#x0A5C]
                        | #x0A5E | [#x0A72-#x0A74] | [#x0A85-
                        #x0A8B] | #x0A8D | [#x0A8F-#x0A91]
                        | [#x0A93-#x0AA8] | [#x0AAA-#x0AB0]
                        | [#x0AB2-#x0AB3] | [#x0AB5-#x0AB9]
                        | #x0ABD | #x0AE0 | [#x0B05-#x0B0C]
                        | [#x0B0F-#x0B10] | [#x0B13-#x0B28]
                        | [#x0B2A-#x0B30] | [#x0B32-#x0B33]
                        | [#x0B36-#x0B39] | #x0B3D | [#x0B5C-
                        #x0B5D] | [#x0B5F-#x0B61] | [#x0B85-
                        #x0B8A] | [#x0B8E-#x0B90] | [#x0B92-
                        #x0B95] | [#x0B99-#x0B9A] | #x0B9C
                        | [#x0B9E-#x0B9F] | [#x0BA3-#x0BA4]
                        | [#x0BA8-#x0BAA] | [#x0BAE-#x0BB5]
                        | [#x0BB7-#x0BB9] | [#x0C05-#x0C0C]
                        | [#x0C0E-#x0C10] | [#x0C12-#x0C28]
                        | [#x0C2A-#x0C33] | [#x0C35-#x0C39]
                        | [#x0C60-#x0C61] | [#x0C85-#x0C8C]
                        | [#x0C8E-#x0C90] | [#x0C92-#x0CA8]
                        | [#x0CAA-#x0CB3] | [#x0CB5-#x0CB9]
                        | #x0CDE | [#x0CE0-#x0CE1] | [#x0D05-
                        #x0D0C] | [#x0D0E-#x0D10] | [#x0D12-
                        #x0D28] | [#x0D2A-#x0D39] | [#x0D60-
                        #x0D61] | [#x0E01-#x0E2E] | #x0E30
                        | [#x0E32-#x0E33] | [#x0E40-#x0E45]
                        | [#x0E81-#x0E82] | #x0E84 | [#x0E87-
                        #x0E88] | #x0E8A | #x0E8D | [#x0E94-
                        #x0E97] | [#x0E99-#x0E9F] | [#x0EA1-
                        #x0EA3] | #x0EA5 | #x0EA7 | [#x0EAA-
                        #x0EAB] | [#x0EAD-#x0EAE] | #x0EB0
                        | [#x0EB2-#x0EB3] | #x0EBD | [#x0EC0-
                        #x0EC4] | [#x0F40-#x0F47] | [#x0F49-
                        #x0F69] | [#x10A0-#x10C5] | [#x10D0-
                        #x10F6] | #x1100 | [#x1102-#x1103]
                        | [#x1105-#x1107] | #x1109 | [#x110B-
                        #x110C] | [#x110E-#x1112] | #x113C
                        | #x113E | #x1140 | #x114C | #x114E
                        | #x1150 | [#x1154-#x1155] | #x1159
                        | [#x115F-#x1161] | #x1163 | #x1165
                        | #x1167 | #x1169 | [#x116D-#x116E]
                        | [#x1172-#x1173] | #x1175 | #x119E
```

```
                              | #x11A8 | #x11AB | [#x11AE-#x11AF]
                              | [#x11B7-#x11B8] | #x11BA | [#x11BC-
                              #x11C2] | #x11EB | #x11F0 | #x11F9
                              | [#x1E00-#x1E9B] | [#x1EA0-#x1EF9]
                              | [#x1F00-#x1F15] | [#x1F18-#x1F1D]
                              | [#x1F20-#x1F45] | [#x1F48-#x1F4D]
                              | [#x1F50-#x1F57] | #x1F59 | #x1F5B
                              | #x1F5D | [#x1F5F-#x1F7D] | [#x1F80-
                              #x1FB4] | [#x1FB6-#x1FBC] | #x1FBE
                              | [#x1FC2-#x1FC4] | [#x1FC6-#x1FCC]
                              | [#x1FD0-#x1FD3] | [#x1FD6-#x1FDB]
                              | [#x1FE0-#x1FEC] | [#x1FF2-#x1FF4]
                              | [#x1FF6-#x1FFC] | #x2126 | [#x212A-
                              #x212B] | #x212E | [#x2180-#x2182]
                              | [#x3041-#x3094] | [#x30A1-#x30FA]
                              | [#x3105-#x312C] | [#xAC00-#xD7A3]
Ideographic             ::= [#x4E00-#x9FA5] | #x3007 | [#x3021-
                              #x3029]
CombiningChar           ::= [#x0300-#x0345] | [#x0360-#x0361]
                              | [#x0483-#x0486] | [#x0591-#x05A1]
                              | [#x05A3-#x05B9] | [#x05BB-#x05BD]
                              | #x05BF | [#x05C1-#x05C2] | #x05C4
                              | [#x064B-#x0652] | #x0670 | [#x06D6-
                              #x06DC] | [#x06DD-#x06DF] | [#x06E0-
                              #x06E4] | [#x06E7-#x06E8] | [#x06EA-
                              #x06ED] | [#x0901-#x0903] | #x093C
                              | [#x093E-#x094C] | #x094D | [#x0951-
                              #x0954] | [#x0962-#x0963] | [#x0981-
                              #x0983] | #x09BC | #x09BE | #x09BF
                              | [#x09C0-#x09C4] | [#x09C7-#x09C8]
                              | [#x09CB-#x09CD] | #x09D7 | [#x09E2-
                              #x09E3] | #x0A02 | #x0A3C | #x0A3E
                              | #x0A3F | [#x0A40-#x0A42] | [#x0A47-
                              #x0A48] | [#x0A4B-#x0A4D] | [#x0A70-
                              #x0A71] | [#x0A81-#x0A83] | #x0ABC
                              | [#x0ABE-#x0AC5] | [#x0AC7-#x0AC9]
                              | [#x0ACB-#x0ACD] | [#x0B01-#x0B03]
                              | #x0B3C | [#x0B3E-#x0B43] | [#x0B47-
                              #x0B48] | [#x0B4B-#x0B4D] | [#x0B56-
                              #x0B57] | [#x0B82-#x0B83] | [#x0BBE-
                              #x0BC2] | [#x0BC6-#x0BC8] | [#x0BCA-
                              #x0BCD] | #x0BD7 | [#x0C01-#x0C03]
                              | [#x0C3E-#x0C44] | [#x0C46-#x0C48]
                              | [#x0C4A-#x0C4D] | [#x0C55-#x0C56]
                              | [#x0C82-#x0C83] | [#x0CBE-#x0CC4]
                              | [#x0CC6-#x0CC8] | [#x0CCA-#x0CCD]
                              | [#x0CD5-#x0CD6] | [#x0D02-#x0D03]
                              | [#x0D3E-#x0D43] | [#x0D46-#x0D48]
                              | [#x0D4A-#x0D4D] | #x0D57 | #x0E31
                              | [#x0E34-#x0E3A] | [#x0E47-#x0E4E]
                              | #x0EB1 | [#x0EB4-#x0EB9] | [#x0EBB-
                              #x0EBC] | [#x0EC8-#x0ECD] | [#x0F18-
                              #x0F19] | #x0F35 | #x0F37 | #x0F39
                              | #x0F3E | #x0F3F | [#x0F71-#x0F84]
                              | [#x0F86-#x0F8B] | [#x0F90-#x0F95]
                              | #x0F97 | [#x0F99-#x0FAD] | [#x0FB1-
                              #x0FB7] | #x0FB9 | [#x20D0-#x20DC]
```

```
                              | #x20E1 | [#x302A-#x302F] | #x3099
                              | #x309A
CombiningChar             ::= [#x0300-#x0345] | [#x0360-#x0361]
                              | [#x0483-#x0486] | [#x0591-#x05A1]
                              | [#x05A3-#x05B9] | [#x05BB-#x05BD]
                              | #x05BF | [#x05C1-#x05C2] | #x05C4
                              | [#x064B-#x0652] | #x0670 | [#x06D6-
                              #x06DC] | [#x06DD-#x06DF] | [#x06E0-
                              #x06E4] | [#x06E7-#x06E8] | [#x06EA-
                              #x06ED] | [#x0901-#x0903] | #x093C
                              | [#x093E-#x094C] | #x094D | [#x0951-
                              #x0954] | [#x0962-#x0963] | [#x0981-
                              #x0983] | #x09BC | #x09BE | #x09BF
                              | [#x09C0-#x09C4] | [#x09C7-#x09C8]
                              | [#x09CB-#x09CD] | #x09D7 | [#x09E2-
                              #x09E3] | #x0A02 | #x0A3C | #x0A3E
                              | #x0A3F | [#x0A40-#x0A42] | [#x0A47-
                              #x0A48] | [#x0A4B-#x0A4D] | [#x0A70-
                              #x0A71] | [#x0A81-#x0A83] | #x0ABC
                              | [#x0ABE-#x0AC5] | [#x0AC7-#x0AC9]
                              | [#x0ACB-#x0ACD] | [#x0B01-#x0B03]
                              | #x0B3C | [#x0B3E-#x0B43] | [#x0B47-
                              #x0B48] | [#x0B4B-#x0B4D] | [#x0B56-
                              #x0B57] | [#x0B82-#x0B83] | [#x0BBE-
                              #x0BC2] | [#x0BC6-#x0BC8] | [#x0BCA-
                              #x0BCD] | #x0BD7 | [#x0C01-#x0C03]
                              | [#x0C3E-#x0C44] | [#x0C46-#x0C48]
                              | [#x0C4A-#x0C4D] | [#x0C55-#x0C56]
                              | [#x0C82-#x0C83] | [#x0CBE-#x0CC4]
                              | [#x0CC6-#x0CC8] | [#x0CCA-#x0CCD]
                              | [#x0CD5-#x0CD6] | [#x0D02-#x0D03]
                              | [#x0D3E-#x0D43] | [#x0D46-#x0D48]
                              | [#x0D4A-#x0D4D] | #x0D57 | #x0E31
                              | [#x0E34-#x0E3A] | [#x0E47-#x0E4E]
                              | #x0EB1 | [#x0EB4-#x0EB9] | [#x0EBB-
                              #x0EBC] | [#x0EC8-#x0ECD] | [#x0F18-
                              #x0F19] | #x0F35 | #x0F37 | #x0F39
                              | #x0F3E | #x0F3F | [#x0F71-#x0F84]
                              | [#x0F86-#x0F8B] | [#x0F90-#x0F95]
                              | #x0F97 | [#x0F99-#x0FAD] | [#x0FB1-
                              #x0FB7] | #x0FB9 | [#x20D0-#x20DC]
                              | #x20E1 | [#x302A-#x302F] | #x3099
                              | #x309A
Extender                  ::= #x00B7 | #x02D0 | #x02D1 | #x0387
                              | #x0640 | #x0E46 | #x0EC6 | #x3005
                              | [#x3031-#x3035] | [#x309D-#x309E]
                              | [#x30FC-#x30FE]
```

Appendix C – XQuery 1.0: An XML Query Language

W3C Working Draft 30 April 2002

This version:
http://www.w3.org/TR/2002/WD-xquery-20020430

Latest version:
http://www.w3.org/TR/xquery

Previous versions:
http://www.w3.org/TR/2001/WD-xquery-20011220 http://www.w3.org/TR/2001/WD-xquery-20010607

Editors:
Scott Boag (XSL WG), IBM Research <scott_boag@us.ibm.com>
Don Chamberlin (XML Query WG), IBM Almaden Research Center <chamberlin@almaden.ibm.com>
Mary F. Fernandez (XML Query WG), AT&T Labs <mff@research.att.com>
Daniela Florescu (XML Query WG), XQRL <dana@xqrl.com>
Jonathan Robie (XML Query WG), Invited Expert <jonathan.robie@datadirect-technologies.com>
Jérôme Siméon (XML Query WG), Bell Labs, Lucent Technologies <simeon@research.bell-labs.com>
Mugur Stefanescu (XML Query WG), Concentric Visions <MStefanescu@Concentricvisions.com>

Abstract

XML is a versatile markup language, capable of labeling the information content of diverse data sources including structured and semi-structured documents, relational databases, and object repositories. A query language that uses the structure of XML intelligently can express queries across all these kinds of data, whether physically stored in XML or viewed as XML via middleware. This specification describes a query language called XQuery, which is designed to be broadly applicable across many types of XML data sources.

Status of this Document

This is a public W3C Working Draft for review by W3C Members and other interested parties. This section describes the status of this document at the time of its publication. It is a draft document and may be updated, replaced, or made obsolete by other documents at any time. It is inappropriate to use W3C Working Drafts as reference material or to cite them as other than

"work in progress." A list of current public W3C technical reports can be found at http://www.w3.org/TR/.

Much of this document is the result of joint work by the XML Query and XSL Working Groups, which are jointly responsible for XPath 2.0, a language derived from both XPath 1.0 and XQuery. The XPath 2.0 and XQuery 1.0 Working Drafts are generated from a common source. These languages are closely related, sharing much of the same expression syntax and semantics, and much of the text found in the two Working Drafts is identical.

This version of the document contains new details about the type system of XQuery, including a syntax for declaring types in function signatures and other expressions. It describes the semantics of several expressions that operate on types, including `treat`, `assert`, and `validate` expressions. It also describes in greater detail the semantics of element and attribute constructors and how they operate on the underlying data model.

This document is a work in progress. It contains many open issues, and should not be considered to be fully stable. Vendors who wish to create preview implementations based on this document do so at their own risk. While this document reflects the general consensus of the working groups, there are still controversial areas that may be subject to change.

Public comments on this document and its open issues are welcome. Of particular interest are comments on error handling (see issues 97 and 98.) Comments should be sent to the W3C XPath/XQuery mailing list, public-qt-comments@w3.org (archived at http://lists.w3.org/Archives/Public/public-qt-comments/).

XQuery 1.0 has been defined jointly by the XML Query Working Group (part of the XML Activity) and the XSL Working Group (part of the Style Activity).