XQuark Fusion 1.1

XQuery Reference Guide

XQUARK FUSION 1.1

XQUERY REFERENCE GUIDE

Document version 1.1

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 12 November 2003. See http://www.w3.org/TR/2003/WD-xquery-20031112

XQuery 1.0 and XPath 2.0 Functions and Operators. W3C Working Draft 12 November 2003. See http://www.w3.org/TR/2003/WD-xpath-functions-20031112/

XQuery 1.0 and XPath 2.0 Data Model. W3C Working Draft 12 November 2003. See http://www.w3.org/TR/2003/WD-xpath-datamodel-20031112/

XML Query Use Cases. W3C Working Draft 12 November 2003. See http://www.w3.org/TR/2003/WD-xquery-use-cases-20031112/

XQuery 1.0 Formal Semantics. W3C Working Draft 12 November 2003. See http://www.w3.org/TR/2003/WD-xquery-semantics-20031112/

XSLT 2.0 and XQuery 1.0 Serialization. W3C Working Draft 12 November 2003. See http://www.w3.org/TR/2003/WD-xslt-xquery-serialization-20031112/

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

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 Fusion provides an implementation compatible with XQuery and applicable to heterogeneous data distributed accross the enterprise. It is a middleware that wraps multiple and heterogeneous data sources into collections of XML documents, which can then be queried using XQuery.

This document is the query reference guide for XQuark Fusion: it describes valid expressions of the language, as well as the specific 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. 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 **November 12, 2003**, 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 Fusion provides an implementation compatible with XQuery and applicable to various data sources (including XML documents and relational databases). It does so by defining views on top of those sources, and by querying these views using XQuery expressions and built-in functions.

XQuark Fusion provides mechanisms to:

- Expose several heterogeneous data sources as a single, virtual XML database (made of collections of XML documents). This XML view exports its metadata information as a strongly-typed XML schema.
- Execute strongly-types XQuery queries over the exposed XML views.
- Return query results as newly constructed XML documents.

The above mechanisms represent a complete framework for efficiently publishing enterprise information in XML.

This reference guide is organized in three main sections:

- A description of the XML data model which is constructed by XQuark Fusion from the underlying data sources, and which defines the XML information available to the query processor.

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

- A guide to the XQuery language expressions, or more precisely to the subset of the XQuery expressions which is used by XQuark Fusion to query data sources.
- 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 Fusion 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).

```
<accessor
....name = xs:string
    type = mediator>
....Content: (launcher, specific, subaccessors?)
</accessor>
```

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

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 Fusion, given in Appendix B – Complete BNF Grammar.

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 12 November 2003. See http://www.w3.org/TR/2003/WD-xquery-20031112

[XQuery 1.0 and XPath 2.0 Functions and Operators]
XQuery 1.0 and XPath 2.0 Functions and Operators. W3C Working Draft 12 November 2003. See http://www.w3.org/TR/2003/WD-xpath-functions-20031112/

[XQuery 1.0 and XPath 2.0 Data Model]
XQuery 1.0 and XPath 2.0 Data Model. W3C Working Draft 12 November 2003. See http://www.w3.org/TR/2003/WD-xpath-datamodel-20031112/

[XML Query Use Cases] *XML Query Use Cases*. W3C Working Draft 12 November 2003. See http://www.w3.org/TR/2003/WD-xquery-use-cases-20031112/

[XQuery 1.0 Formal Semantics]
XQuery 1.0 Formal Semantics. W3C Working Draft 12 November 2003. See http://www.w3.org/TR/2003/WD-xquery-semantics-20031112/

[XSLT 2.0 and XQuery 1.0 Serialization]
XSLT 2.0 and XQuery 1.0 Serialization. W3C Working Draft 12 November 2003. See http://www.w3.org/TR/2003/WD-xslt-xquery-serialization-20031112/

[XQuark Bridge XQuery Reference Guide]
XQuark Bridge XQuery Reference Guide. See http://www.xquark.org/bridge/guide/xquery-reference.pdf

3

XQuery Data Model for XQuark Fusion

This section describes the XML view, defined on top of heterogeneous data sources, which can be queried through XQuark Fusion.

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 a *sequence* of zero or more *items*. 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 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 type of the item is unknown.

Every node has a **string value**, which is the textual content of the node, as in XPath 1.0, and 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, text or processing instruction node is its string value.
- The typed value of an element or attribute node that has no type annotation is its string value.
- 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 not defined, accessing it raises an error.

The XQuark Fusion data model

XQuark Fusion uses the standard XQuery data model. Data model instances are retrieved from two kinds of sources:

- XQuery data sources, or more precisely data sources supporting the XQuark-defined XML/DBC API: this currently includes XQuark Bridge, for accessing relational databases, and XQuark Fusion sources. The number of supported sources will increase once a standard Java API for XQuery (XQJ – JSR 225) will be available, as the XML/DBC API will be aligned with this standard. Data model instances stored in those sources are made available through the `fn:collection` built-in function. Data models instances obtained from collections are usually strongly typed, as XML/DBC data sources can associate metadata information expressed as XML Schema to all their published collections. The syntax that must be used to access data model instances in data source collections is described in Accessing data in data sources.
- XML documents: data model instances can be created from XML documents using the `fn:doc` built-in function. These instances can be either typed or untyped, depending on the availability of an appropriate schema when the document is parsed.

Configuring the XQuark Fusion data sources

As described above, XQuark Fusion can publish the content of several data sources as an integrated XML view, which can then be queried using XQuery.

In order to be accessible by XQuark Fusion, those data sources need to be declared and parameterized in a configuration file. Note that even when accessing only documents, a (almost empty) configuration file must be provided.

Configuration File

The configuration file describes the XQuark Fusion component and its sub components (currently limited to XQuark Bridge or XQuark Fusion components).

The general structure of a configuration file is shown below:

```
<accessor name="{ Component name }"
          type="{ Component type }">
  <launcher type="{ Laucher type }"/>
  <specific>{ Any well-formed XML content}</specific>
  <subaccessors>
    <subaccessor name="{ Datasource name }">
      <driver>{ XML/DBC driver name }</driver>
```

```
      <connection>{ XML/DBC connection URL }</connection>
    </subaccessor>
    …
  </subaccessors>
</accessor>
```

The complete XML Schema for the configuration file is given in Appendix A - XML Schema for the XQuark Fusion configuration file. This schema is associated to the namespace identified by the following URI: `http://www.xquark.org/Mediator`.

Two main sections appear in the configuration file:

- The declarations related to the main XQuark Fusion component itself,
- The declarations of the subaccessors, i.e. data sources to be accessed by the component.

The first section is composed of the following elements:

```
<accessor
....name = xs:string
    type = mediator>
....Content: (launcher, specific, subaccessors?)
</accessor>

<launcher
    type = jvm>
    Content: empty
</launcher>

<specific>
    Content: any
</specific>
```

It contains information on the XQuark Fusion component, as an `accessor` element :

- A mandatory `type` attribute: the only type currently supported is "mediator".
- A mandatory `name` attribute: an identifier defined by the application designer for this XQuark Fusion instance.
- A mandatory launcher `type` attribute: the only type currently supported is "jvm".
- Additional, specific information: not currently used.

Note that even if the `launcher` and `specific` elements do not currently carry out any useful information, they must still appear in the configuration file.

The second section is composed of the following elements:

```
<subaccessors>
....Content: subaccessor*
</subaccessors>

<subaccessor
    name = xs:string>
    Content: (driver?, connection)
</subaccessor>

<driver>
    Content: xs:string
</driver>

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

It contains a wrapper element (`subaccessors`) and information about each accessible data source, in each `subaccessor` element:

- The mandatory `name` attribute contains the name used in the application to refer to the data source. This name is important when referring to collections belonging to the data source.
- The optional `driver` element contains the name of the XML/DBC driver class used to access the data source. Valid values currently include `org.xquark.extractor.ExtractorDriver` and `org.xquark.mediator.MediatorDriver`. Both drivers are pre-loaded in XQuark Fusion, therefore it is not necessary to specify this information.
- The mandatory `connection` element contains the XML/DBC connection string used to access the data source. This string is similar to a JDBC URL, and contains the following components, separated by colons (':'):
  - The protocol name: it must be "xdbc",
  - The vendor name: it must currently be "xquark",
  - The source type: it must be one of "mediator"(XQuark Fusion) or "extractor" (XQuark Bridge),
  - A configuration file URL for the appropriate source type, i.e. either an XQuark Bridge configuration file (as described in [XQuark Bridge XQuery Reference Guide]), containing among other things the JDBC connection parameters for the data source, or another XQuark Fusion configuration file. Both absolute and relative URLs can be used.

  For instance, "xdbc:xquark:mediator:file:med-conf/main.xml" is a valid connection string for XQuark Fusion.

Accessing data in data sources

Once configured, XQuark Fusion data sources expose two kinds of metadata information to the XQuery processor:

- Collection names: they are of the form `<data source name>:<local collection name>`, where `<data source name>` is the name of the data source, as declared in the XQuark Fusion configuration file, and `<local collection name>` is the name of the collection within the data source itself. In the case of a relational data source, a collection name is often a table name, although the mapping between relational table names and collection names can be customized in the XQuark Bridge configuration file.
- Collection types: data model instances retrieved from relational collections are strongly typed, using an XML Schema generated from the relational metadata. Among other things, the generated schema contains a top-level element declaration for each published relational table, whose name is the name of the corresponding table. By default, the generated XML schema does not have a target namespace. However, when more than one data source are accessed from XQuark Fusion, each XQuark Bridge data source must be properly configured to associate a user-defined target namespace to the generated schema, in order to avoid name conflicts. See the [XQuark Bridge XQuery Reference Guide] for details on how to define the target namespace for the generated schema.

For instance, considering two relational data sources RDB1 and RDB2, containing the following sets of relational tables:

- RDB1: { A, B }
- RDB2: { B, C }

If RDB1 and RDB2 are respectively named S1 and S2 in the XQuark Fusion configuration file, the exposed collection names will be: S1:A, S1:B, S2:B and S2:C.

Assuming that target namespaces "http://RDB1" and "http://RDB2" have been associated respectively to RDB1 and RDB2 in the appropriate XQuark Bridge configuration files, the available top-level element declarations will be:

- A and B in target namespace "http://RDB1" for RDB1,
- B and C in target namespace "http://RDB2" for RDB2.

The following is a valid query that could be executed in an XQuark Fusion instance configured as described above:

```
declare namespace n1 = "http://RDB1"
declare namespace n2 = "http://RDB2"
```

```
for $a in collection("S1:A")/n1:A,
    $c in collection("S2:C")/n2:C
where $a/C1 = $c/C2
return
    <result>{ $a/C3 } {$c/C4}</result>
```

The above query will join tuples from table A in RDB1 and table C in RDB2 on columns A.C1 and C.C2, returning the values contained in columns A.C3 and C.C4.

Wildcards can be used to simplify the above query:

```
for $a in collection("*:A")/*:A,
    $c in collection("*:C")/*:C
where $a/C1 = $c/C2
return
    <result>{ $a/C3 } {$c/C4}</result>
```

The XQuery processor will perform wildcard expansion both in collection names and navigation steps, based on available metadata information. In this example, the result will be the same as for the previous query.

Wildcard expansion can also be used to perform implicit unions over collections with the same local name in several data sources. For instance, the following query returns the content of the columns B.C1 in both data sources (this assumes that B tables in both data sources have a C1 column):

```
for $b in collection("*:B")/*:B
return
    <result>{ $b/C1 }</result>
```

Part

4

XQuery queries

Language specification reference

The XQuery Language specification implemented by XQuark Fusion is available at http://www.w3.org/TR/2003/WD-xquery-20031112/.

XQuark Fusion only implements a subset of the above draft specification. Rather than repeating the specification contents, this chapter describes the main characteristics of the XQuery Language that are **not** supported by the product.

Grammar rules are referenced by their number as defined in the specification of the XQuery Language. Unsupported grammar elements are shown in bold. Partially supported grammar elements (i.e. those supported with some restrictions) are shown in italics. Extension grammar elements defined by XQuark are underlined.

All grammar rules can be found in Appendix B.

Unsupported or partially supported features

Prolog declarations

Variable declarations

External variable declarations are not supported.

```
[38] VarDecl ::= <"declare" "variable" "$"> VarName
                 TypeDeclaration? (("{" Expr "}") |
                 "external")
```

Function declarations

External function declarations are not supported.

```
[120] FunctionDecl     <"declare" "function"> <QName "("> 
                       ParamList? (")" | (<")" "as">
                       SequenceType)) (EnclosedExpr |
                       "external")
```

Functions are always inlined within the expression in which they are called: they are in fact considered as **parameterized views** by the XQuery parser. This approach induces a few restrictions on the supported expressions within a function body:

- Recursive functions are not supported, even in the case when the recursion is indirect, i.e. when there is a cyclic function call graph.

- Functions should not return expressions that cannot be handled in the `where` clause of a FLWOR. For instance, conditional expressions or computed element constructors should not be used in function return values.

Path expressions

Filter steps

Filter steps are only supported as the first step in a path expression. Furthermore, predicates are not allowed in a filter step.

```
[71] StepExpr   ::= AxisStep | FilterStep
[73] FilterStep ::= PrimaryExpr Predicates
```

Forward axes

The `following` and `following-sibling` forward axes are not supported.

```
[89] ForwardAxis ::= <"child" "::">
                   | <"descendant" "::">
                   | <"attribute" "::">
                   | <"self" "::">
                   | <"descendant-or-self" "::">
                   | <"following-sibling" "::">
                   | <"following" "::">
```

Reverse axes

The only supported reverse axis is `parent`.

```
[90] ReverseAxis ::= <"parent" "::">
                   | <"ancestor" "::">
                   | <"preceding-sibling" "::">
                   | <"preceding" "::">
                   | <"ancestor-or-self" "::">
```

Predicates

The XPath concepts of context position (position of the context item, i.e. the current node, in the current nodeset) and context size (size of the current nodeset) are not supported. Thus, predicate expressions returning numeric values are not supported either.

```
[77] Predicates     ("[" Expr "]")*
```

Kind tests

Only the `text()` and `node()` kind tests are supported in step expressions. Note however that all kind tests can by used in sequence type expressions.

```
[128] KindTest     DocumentTest
                   | ElementTest
                   | AttributeTest
                   | PITest
                   | CommentTest
                   | TextTest
                   | AnyKindTest
```

Sequence expressions

Sequence constructors

Sequence construction can only be used for literals, within the `where` clause of FLWOR expressions.

```
[40] Expr ::= ExprSingle ("," ExprSingle)*
```

Range expressions

Construction of a sequence of values using a range expression is not supported.

```
[62] RangeExpr     AdditiveExpr ( "to" AdditiveExpr )?
```

Set operations

Union, intersection and difference of sequence expressions are not supported.

```
[66] UnionExpr           ::= IntersectExceptExpr ( ("union"
                             | "|") IntersectExceptExpr )*
[67] IntersectExceptExpr    ValueExpr ( ("intersect" |
                            "except") ValueExpr )*
```

Arithmetic expressions

Integral division

The `idiv` operator is not supported.

```
[64] MultiplicativeExpr ::= UnaryExpr ( ("*" | "div" |
                            "idiv" | "mod") UnaryExpr )*
```

Comparison expressions

Node comparison

Node comparison is not supported.

```
[61] ComparisonExpr     RangeExpr ( (ValueComp
                        | GeneralComp
                        | NodeComp) RangeExpr )?
[84] NodeComp       ::= "is" | "<<" | ">>"
```

Constructors

Computed constructors

Computed constructors are only supported in the return clause of a FLWOR expression of the main module.

```
[81] ComputedConstructor ::= CompElemConstructor
                           | CompAttrConstructor
                           | CompDocConstructor
                           | CompTextConstructor
                           | CompXmlPI
                           | CompXmlComment
                           | CompNSConstructor
```

FLWOR expressions

Positional variable

Positional variables are not available.

```
[43] ForClause      ::= <"for" "$"> VarName TypeDeclaration?
                        PositionalVar? "in" ExprSingle ("," 
                        "$" VarName TypeDeclaration?
                        PositionalVar? "in" ExprSingle)*
[44] PositionalVar      "at" "$" VarName
```

Stable order

Stable order is not supported.

```
[47] OrderByClause    (<"order" "by"> | <"stable" "order"
                      "by">) OrderSpecList
```

Order modifier

Only the `ascending` and `descending` modifiers are supported.

```
[50] OrderModifier    ("ascending" | "descending")?
                      (<"empty" "greatest"> | <"empty"
                      "least">)? ("collation"
                      StringLiteral)?
```

Hints

XQuark Fusion extends standard FLWOR expressions by providing hints, using the standard **pragma** extension mechanism. Hint clauses can be added before the `where` clause of a FLWOR expression to better control the join order and algorithms chosen by Fusion when executing a distributed query.

```
[42]    FLWORExpr    ::=  (ForClause | LetClause)+
                          HintClause? WhereClause?
                          OrderByClause? "return" ExprSingle
[ext1]  HintClause   ::=  <"(::" "pragma" "hint">
                          ("outer-nested" | "outer-merge")?
                          NodeClause? "::)"
[ext2]  NodeClause   ::=  ("merge" | "nested-loop")
                          "(" LeafClause "," LeafClause ")"
[ext3]  LeafClause   ::=  NodeClause | "$" VarName
```

A hint clause contains information on how variables of the FLWOR expression are to be joined with each other and also with the variables of the parent FLWOR expression (the FLWOR expression whose `return` clause contains this FLWOR expression). The hint clause contains two parts:

- The optional outer part controls how an inner FLWOR is joined to its enclosing FLWOR. This part uses two keywords, `outer-nested` and `outer-merge`, to specify respectively that the outer join should be performed as a nested loop or as a sorted merge join. The sorted merge join is the default.
- The optional inner part controls how variables in the FLWOR are joined together. As more than two variables can appear in a FLWOR, the hint inner part is represented as a tree, each leaf being a variable and each node being a hint on how to join two nodes (i.e. the result of a previous

join) or leaves. The possible hints are `nested-loop` and `merge`, to trigger respectively a nested loop or a sorted merge join. Note that all FLWOR variables do not need to appear in the hint inner part. Unspecified variables will be joined to the result of the joins specified by the hint using the default algorithm, namely sorted merge join.

Hints are only used whenever possible, they are ignored if they induce an error. The use of hints might induce additional sort steps after performing joins, in order to restore the original natural order of the FLWOR expression.

Conditional expressions

Conditional (If-Then-Else) expressions can only be used in the return clause of a FLWOR expression of the main module. The condition expression should involve only simple path expressions starting with a variable, literals and built-in functions using the same type of expressions as arguments. Expressions used in the `then` and `else` clauses should not contain nested sub-queries.

```
[54] IfExpr ::= <"if" "("> Expr ")" "then" ExprSingle "else"
                ExprSingle
```

Expressions on Sequence Types

Typeswitch

Type switch expressions are not supported.

```
[41] ExprSingle     ::= FLWORExpr
                        | QuantifiedExpr
                        | TypeswitchExpr
                        | IfExpr
                        | OrExpr
[52] TypeswitchExpr     <"typeswitch" "("> Expr ")"
                        CaseClause+ "default" ("$" VarName)?
                        "return" ExprSingle
[53] CaseClause     ::= "case" ("$" VarName "as")?
                        SequenceType "return" ExprSingle
```

Instance of

The expression type cannot be tested.

```
[57] InstanceofExpr     TreatExpr ( <"instance" "of">
                        SequenceType )?
```

Casting operations

Static and dynamic casting is not supported.

```
[58] TreatExpr    ::= CastableExpr ( <"treat" "as">
                      SequenceType )?
[59] CastableExpr     CastExpr ( <"castable" "as">
                      SingleType )?
[60] CastExpr     ::= ComparisonExpr ( <"cast" "as">
                      SingleType )?
```

Validate expressions

Explicit validation expressions are not supported. Note however that implicit validation, based on the global validation mode, in-scope schema definitions and contructed element QNames, is fully supported.

```
[68] ValueExpr     ::= ValidateExpr | PathExpr
[78] ValidateExpr  ::= (<"validate" "{"> | (<"validate"
                       "global"> "{") | (<"validate"
                       "context"> SchemaContextLoc "{") |
                       (<"validate" SchemaMode>
                       SchemaContext? "{")) Expr "}"
[79] SchemaContext     ("context" SchemaContextLoc) |
                       "global"
```

Part

5

Supported built-in functions

Supported build-in functions

The following functions are supported by XQuark Fusion:

- fn:abs
- fn:avg
- fn:ceiling
- fn:collection
- fn:concat
- fn:contains
- fn:count
- fn:current-date
- fn:current-datetime
- fn:current-time
- fn:data
- fn:deep-equals
- fn:distinct-values
- fn:empty
- fn:ends-with
- fn:exists
- fn:false
- fn:floor
- fn:matches
- fn:max
- fn:min
- fn:not

- fn:number
- fn:round
- fn:starts-with
- fn:string-length
- fn:substring
- fn:sum
- fn:true
- fn:upper-case

Note that the `fn:doc` function is not supported, as XQuark Fusion can only access collections stored in a relational database.

Supported built-in type constructors

The following type constructors are supported by XQuark Fusion:

- xs:date
- xs:datetime
- xs:decimal
- xs:double
- xs:float
- xs:integer
- xs:string
- xs:time

Note that those constructors can only take string literals as parameters.

Index

Appendix A – XML Schema for the XQuark Fusion configuration file

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

  <element name="accessor">
    <complexType>
      <sequence>
        <element ref="med:launcher"/>
        <element ref="med:specific"/>
        <element ref="med:subaccessors" minOccurs="0" />
      </sequence>
      <attribute name="type" type="string" use="required"/>
      <attribute name="name" type="string" use="required"/>
    </complexType>
  </element>

  <element name="classname">
    <complexType mixed="true"/>
  </element>

  <element name="conffile">
    <complexType mixed="true" />
  </element>

  <element name="host">
    <complexType mixed="true" />
  </element>

  <element name="url">
    <complexType mixed="true" />
  </element>

  <element name="object">
    <complexType mixed="true" />
  </element

  <element name="port">
    <complexType mixed="true" />
  </element>

  <element name="launcher">
    <complexType>
      <sequence>
        <element ref="med:classname" minOccurs="0">
        <element ref="med:conffile" minOccurs="0"/>
        <element ref="med:host" minOccurs="0"/>
        <element ref="med:port" minOccurs="0"/>
        <element ref="med:object" minOccurs="0"/>
        <element ref="med:url" minOccurs="0"/>
      </sequence>
      <attribute name="type" type="string" use="required"/>
```

```
    </complexType>
  </element>

  <element name="specific">
    <complexType>
      <sequence>
        <any namespace="##any" minOccurs="0"
             maxOccurs="unbounded"/>
      </sequence>
    </complexType>
  </element>

  <element name="driver">
    <complexType mixed="true" />
  </element>

  <element name="connection">
    <complexType mixed="true" />
  </element>

  <element name="user">
    <complexType mixed="true" />
  </element>

  <element name="password">
    <complexType mixed="true" />
  </element>

  <element name="subaccessor">
    <complexType>
      <sequence>
        <element ref="med:driver" minOccurs="0" />
        <element ref="med:connection" minOccurs="0" />
        <element ref="med:user" minOccurs="0" />
        <element ref="med:password" minOccurs="0" />
      </sequence>
      <attribute name="name" type="string" use="required"/>
    </complexType>
  </element>

  <element name="subaccessors">
    <complexType>
      <sequence>
        <element ref="med:subaccessor" minOccurs="0"
                 maxOccurs="unbounded" />
      </sequence>
    </complexType>
  </element>

</schema>
```

Appendix B – Complete BNF Grammar

Note 1 : Rule numbers are those used in XQuery language specifications.
Note 2 : Bold items are not supported in the current version. Items in italics are only partially supported. Underlined items are XQuark-defined extensions.

Named Terminals

```
[1]  Pragma                ::= "(::" "pragma" QName PragmaContents* "::)"
[2]  MUExtension           ::= "(::" "extension" QName ExtensionContents*
                               "::)"
[3]  ExprComment           ::= "(:" (ExprCommentContent | ExprComment)* ":)"
[4]  ExprCommentContent    ::= Char
[5]  PragmaContents        ::= Char
[6]  ExtensionContents     ::= Char
[7]  IntegerLiteral        ::= Digits
[8]  DecimalLiteral        ::= ("." Digits) | (Digits "." [0-9]*)
[9]  DoubleLiteral         ::= (("." Digits) | (Digits ("." [0-9]*)?)) ("e" |
                               "E") ("+" | "-")? Digits
[10] StringLiteral         ::= ('"' (PredefinedEntityRef | CharRef | ('"'
                               '"') | [^"&])* '"') | ("'"
                               (PredefinedEntityRef | CharRef | ("'" "'") |
                               [^'&])* "'")
[11] S                     ::= [http://www.w3.org/TR/REC-xml#NT-S] XML
[12] SchemaMode            ::= "lax" | "strict" | "skip"
[13] SchemaGlobalTypeName  ::= "type" "(" QName ")"
[14] SchemaGlobalContext   ::= QName | SchemaGlobalTypeName
[15] SchemaContextStep     ::= QName
[16] Digits                ::= [0-9]+
[17] EscapeQuot            ::= '"' '"'
[18] PITarget              ::= NCName
[19] NCName                ::= [http://www.w3.org/TR/REC-xml-names/#NT-
                               NCName] Names
[20] VarName               ::= QName
[21] Qname                 ::= [http://www.w3.org/TR/REC-xml-names/#NT-QName]
                               Names
[22] PredefinedEntityRef   ::= "&" ("lt" | "gt" | "amp" | "quot" | "apos")
                               ";"
[23] HexDigits             ::= ([0-9] | [a-f] | [A-F])+
[24] CharRef               ::= "&#" (Digits | ("x" HexDigits)) ";"
[25] EscapeApos            ::= "''"
[26] Char                  ::= [http://www.w3.org/TR/REC-xml#NT-Char] XML
[27] ElementContentChar    ::= Char - [{}<&]
[28] QuotAttContentChar    ::= Char - ["{}<&]
[29] AposAttContentChar    ::= Char - ['{}<&]
```

Non-Terminals

```
[30]  Module              ::= VersionDecl? (MainModule | LibraryModule)
[31]  MainModule          ::= Prolog QueryBody
[32]  LibraryModule       ::= ModuleDecl Prolog
[33]  ModuleDecl          ::= <"module" "namespace"> NCName "="
                              StringLiteral Separator
[34]  Prolog              ::= ((NamespaceDecl
                              | XMLSpaceDecl
                              | DefaultNamespaceDecl
                              | DefaultCollationDecl
                              | BaseURIDecl
                              | SchemaImport
                              | ModuleImport
                              | VarDecl
                              | ValidationDecl
                              | FunctionDecl) Separator)*
[35]  Separator           ::= ";"
[36]  VersionDecl         ::= <"xquery" "version" StringLiteral> Separator
[37]  ModuleImport        ::= <"import" "module"> ("namespace" NCName
                              "=")? StringLiteral <"at" StringLiteral>?
[38]  VarDecl             ::= <"declare" "variable" "$"> VarName
                              TypeDeclaration? (("{" Expr "}") |
                              "external")
[39]  QueryBody           ::= Expr
[40]  Expr                ::= ExprSingle ("," ExprSingle)*
[41]  ExprSingle          ::= FLWORExpr
                              | QuantifiedExpr
                              | TypeswitchExpr
                              | IfExpr
                              | OrExpr
[42]  FLWORExpr           ::= (ForClause | LetClause)+ HintClause?
                              WhereClause? OrderByClause? "return"
                              ExprSingle
[43]  ForClause           ::= <"for" "$"> VarName TypeDeclaration?
                              PositionalVar? "in" ExprSingle ("," "$"
                              VarName TypeDeclaration? PositionalVar? "in"
                              ExprSingle)*
[44]  PositionalVar       ::= "at" "$" VarName
[45]  LetClause           ::= <"let" "$"> VarName TypeDeclaration? ":="
                              ExprSingle ("," "$" VarName TypeDeclaration?
                              ":=" ExprSingle)*
[46]  WhereClause         ::= "where" Expr
[47]  OrderByClause       ::= (<"order" "by"> | <"stable" "order" "by">)
                              OrderSpecList
[48]  OrderSpecList       ::= OrderSpec ("," OrderSpec)*
[49]  OrderSpec           ::= ExprSingle OrderModifier
[50]  OrderModifier       ::= ("ascending" | "descending")? (<"empty"
```

```
                                    "greatest"> | <"empty" "least">)?
                                    ("collation" StringLiteral)?
[51]  QuantifiedExpr          ::= (<"some" "$"> | <"every" "$">) VarName
                                  TypeDeclaration? "in" ExprSingle ("," "$"
                                  VarName TypeDeclaration? "in" ExprSingle)*
                                  "satisfies" ExprSingle
[52]  TypeswitchExpr          ::= <"typeswitch" "("> Expr ")" CaseClause+
                                  "default" ("$" VarName)? "return" ExprSingle
[53]  CaseClause              ::= "case" ("$" VarName "as")? SequenceType
                                  "return" ExprSingle
[54]  IfExpr                  ::= <"if" "("> Expr ")" "then" ExprSingle "else"
                                  ExprSingle
[55]  OrExpr                  ::= AndExpr ( "or" AndExpr )*
[56]  AndExpr                 ::= InstanceofExpr ( "and" InstanceofExpr )*
[57]  InstanceofExpr          ::= TreatExpr ( <"instance" "of"> SequenceType
                                  )?
[58]  TreatExpr               ::= CastableExpr ( <"treat" "as"> SequenceType
                                  )?
[59]  CastableExpr            ::= CastExpr ( <"castable" "as"> SingleType )?
[60]  CastExpr                ::= ComparisonExpr ( <"cast" "as"> SingleType )?
[61]  ComparisonExpr          ::= RangeExpr ( (ValueComp
                                  | GeneralComp
                                  | NodeComp) RangeExpr )?
[62]  RangeExpr               ::= AdditiveExpr ( "to" AdditiveExpr )?
[63]  AdditiveExpr            ::= MultiplicativeExpr ( ("+" | "-")
                                  MultiplicativeExpr )*
[64]  MultiplicativeExpr      ::= UnaryExpr ( ("*" | "div" | "idiv" | "mod")
                                  UnaryExpr )*
[65]  UnaryExpr               ::= ("-" | "+")* UnionExpr
[66]  UnionExpr               ::= IntersectExceptExpr ( ("union" | "|")
                                  IntersectExceptExpr )*
[67]  IntersectExceptExpr     ::= ValueExpr ( ("intersect" | "except")
                                  ValueExpr )*
[68]  ValueExpr               ::= ValidateExpr | PathExpr
[69]  PathExpr                ::= ("/" RelativePathExpr?)
                                  | ("//" RelativePathExpr)
                                  | RelativePathExpr
[70]  RelativePathExpr        ::= StepExpr (("/" | "//") StepExpr)*
[71]  StepExpr                ::= AxisStep | FilterStep
[72]  AxisStep                ::= (ForwardStep | ReverseStep) Predicates
[73]  FilterStep              ::= PrimaryExpr Predicates
[74]  ContextItemExpr         ::= "."
[75]  PrimaryExpr             ::= Literal | VarRef | ParenthesizedExpr |
                                  ContextItemExpr | FunctionCall | Constructor
[76]  VarRef                  ::= "$" VarName
[77]  Predicates              ::= ("[" Expr "]")*
```

```
[78]  ValidateExpr          ::= (<"validate" "{"> | (<"validate" "global">
                                "{") | (<"validate" "context">
                                SchemaContextLoc "{") | (<"validate"
                                SchemaMode> SchemaContext? "{")) Expr "}"
[79]  SchemaContext         ::= ("context" SchemaContextLoc) | "global"
[80]  Constructor           ::= DirElemConstructor
                              | ComputedConstructor
                              | XmlComment
                              | XmlPI
                              | CdataSection
[81]  ComputedConstructor   ::= CompElemConstructor
                              | CompAttrConstructor
                              | CompDocConstructor
                              | CompTextConstructor
                              | CompXmlPI
                              | CompXmlComment
                              | CompNSConstructor
[82]  GeneralComp           ::= "=" | "!=" | "<" | "<=" | ">" | ">="
[83]  ValueComp             ::= "eq" | "ne" | "lt" | "le" | "gt" | "ge"
[84]  NodeComp              ::= "is" | "<<" | ">>"
[85]  ForwardStep           ::= (ForwardAxis NodeTest) | AbbrevForwardStep
[86]  ReverseStep           ::= (ReverseAxis NodeTest) | AbbrevReverseStep
[87]  AbbrevForwardStep     ::= "@"? NodeTest
[88]  AbbrevReverseStep     ::= ".."
[89]  ForwardAxis           ::= <"child" "::">
                              | <"descendant" "::">
                              | <"attribute" "::">
                              | <"self" "::">
                              | <"descendant-or-self" "::">
                              | <"following-sibling" "::">
                              | <"following" "::">
[90]  ReverseAxis           ::= <"parent" "::">
                              | <"ancestor" "::">
                              | <"preceding-sibling" "::">
                              | <"preceding" "::">
                              | <"ancestor-or-self" "::">
[91]  NodeTest              ::= KindTest | NameTest
[92]  NameTest              ::= QName | Wildcard
[93]  Wildcard              ::= "*"
                              | <NCName ":" "*">
                              | <"*" ":" NCName>
[94]  Literal               ::= NumericLiteral | StringLiteral
[95]  NumericLiteral        ::= IntegerLiteral | DecimalLiteral |
                                DoubleLiteral
[96]  ParenthesizedExpr     ::= "(" Expr? ")"
[97]  FunctionCall          ::= <QName "("> (ExprSingle ("," ExprSingle)*)?
                                ")"
[98]  DirElemConstructor    ::= "<" QName AttributeList ("/>" | (">"
                                ElementContent* "</" QName S? ">"))
```

```
[99]  CompDocConstructor    ::= <"document" "{"> Expr "}"
[100] CompElemConstructor   ::= (<"element" QName "{"> | (<"element" "{">
                                Expr "}" "{")) Expr? "}"
[101] CompNSConstructor     ::= <"namespace" NCName "{"> Expr "}"
[102] CompAttrConstructor   ::= (<"attribute" QName "{"> | (<"attribute"
                                "{"> Expr "}" "{")) Expr? "}"
[103] CompXmlPI             ::= (<"processing-instruction" NCName "{"> |
                                (<"processing-instruction" "{"> Expr "}"
                                "{")) Expr? "}"
[104] CompXmlComment        ::= <"comment" "{"> Expr "}"
[105] CompTextConstructor   ::= <"text" "{"> Expr? "}"
[106] CdataSection          ::= "<![CDATA[" Char* "]]>"
[107] XmlPI                 ::= "<?" PITarget Char* "?>"
[108] XmlComment            ::= "<!--" Char* "-->"
[109] ElementContent        ::= ElementContentChar
                                | "{{"
                                | "}}"
                                | DirElemConstructor
                                | EnclosedExpr
                                | CdataSection
                                | CharRef
                                | PredefinedEntityRef
                                | XmlComment
                                | XmlPI
[110] AttributeList         ::= (S (QName S? "=" S? AttributeValue)?)*
[111] AttributeValue        ::= ('"' (EscapeQuot | QuotAttrValueContent)*
                                '"')
                                | ("'" (EscapeApos | AposAttrValueContent)*
                                "'")
[112] QuotAttrValueContent  ::= QuotAttContentChar
                                | CharRef
                                | "{{"
                                | "}}"
                                | EnclosedExpr
                                | PredefinedEntityRef
[113] AposAttrValueContent  ::= AposAttContentChar
                                | CharRef
                                | "{{"
                                | "}}"
                                | EnclosedExpr
                                | PredefinedEntityRef
[114] EnclosedExpr          ::= "{" Expr "}"
[115] XMLSpaceDecl          ::= <"declare" "xmlspace"> ("preserve" |
                                "strip")
[116] DefaultCollationDecl  ::= <"declare" "default" "collation">
                                StringLiteral
[117] BaseURIDecl           ::= <"declare" "base-uri"> StringLiteral
[118] NamespaceDecl         ::= <"declare" "namespace"> NCName "="
                                StringLiteral
[119] DefaultNamespaceDecl  ::= (<"declare" "default" "element"> |
```

```
                                <"declare" "default" "function">)
                                "namespace" StringLiteral
[120] FunctionDecl          ::= <"declare" "function"> <QName "(">
                                ParamList? (")" | (<")" "as"> SequenceType))
                                (EnclosedExpr | "external")
[121] ParamList             ::= Param ("," Param)*
[122] Param                 ::= "$" VarName TypeDeclaration?
[123] TypeDeclaration       ::= "as" SequenceType
[124] SingleType            ::= AtomicType "?"?
[125] SequenceType          ::= (ItemType OccurrenceIndicator?)
                                | <"empty" "(" ")">
[126] AtomicType            ::= QName
[127] ItemType              ::= AtomicType | KindTest | <"item" "(" ")">
[128] KindTest              ::= DocumentTest
                                | ElementTest
                                | AttributeTest
                                | PITest
                                | CommentTest
                                | TextTest
                                | AnyKindTest
[129] ElementTest           ::= <"element" "("> ((SchemaContextPath
                                ElementName)
                                | (ElementNameOrWildcard ("("
                                TypeNameOrWildcard "nillable"?)?))? ")"
[130] AttributeTest         ::= <"attribute" "("> ((SchemaContextPath
                                AttributeName)
                                | (AttribNameOrWildcard ("("
                                TypeNameOrWildcard)?))? ")"
[131] ElementName           ::= QName
[132] AttributeName         ::= QName
[133] TypeName              ::= QName
[134] ElementNameOrWildcard ::= ElementName | "*"
[135] AttribNameOrWildcard  ::= AttributeName | "*"
[136] TypeNameOrWildcard    ::= TypeName | "*"
[137] PITest                ::= <"processing-instruction" "("> (NCName |
                                StringLiteral)? ")"
[138] DocumentTest          ::= <"document-node" "("> ElementTest? ")"
[139] CommentTest           ::= <"comment" "("> ")"
[140] TextTest              ::= <"text" "("> ")"
[141] AnyKindTest           ::= <"node" "("> ")"
[142] SchemaContextPath     ::= <SchemaGlobalContext "/"> <SchemaContextStep
                                "/">*
[143] SchemaContextLoc      ::= (SchemaContextPath? QName) |
                                SchemaGlobalTypeName
[144] OccurrenceIndicator   ::= "?" | "*" | "+"
[145] ValidationDecl        ::= <"declare" "validation"> SchemaMode
[146] SchemaImport          ::= <"import" "schema"> SchemaPrefix?
```

```
                                StringLiteral <"at" StringLiteral>?
[147] SchemaPrefix          ::= ("namespace" NCName "=") | (<"default"
                                "element"> "namespace")
```

Additional rules

The following rules have been added to support hints on FLWORs.

```
[ext1] HintClause ::= <"(::" "pragma" "hint">  ("outer-nested" | "outer-
                      merge")? NodeClause "::)"
[ext2] NodeClause ::= ("merge" | "nested-loop") "(" LeafClause "," LeafClause
                      ")"
[ext3] LeafClause ::= NodeClause | "$" VarName
```