

# **Extending XQuery with Collections, Indexes, and Integrity Constraints**

---

---

## Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
<b>2</b>	<b>Overview</b>	<b>2</b>
<b>3</b>	<b>Collections</b>	<b>3</b>
3.1	Declaration . . . . .	3
3.2	Life Cycle Management . . . . .	4
3.3	Accessing and Deleting Collection Data . . . . .	4
<b>4</b>	<b>Indexes</b>	<b>5</b>
4.1	Declaration . . . . .	5
4.2	Life Cycle Management . . . . .	6
4.3	Probing and Maintaining Indexes . . . . .	7
4.3.1	Probing Indexes . . . . .	7
4.3.2	Maintaining Index Data . . . . .	7
<b>5</b>	<b>Integrity Constraints</b>	<b>8</b>
5.1	Declaration . . . . .	8
5.1.1	Entity Integrity . . . . .	8
5.1.2	Domain Integrity . . . . .	8
5.1.3	Referential Integrity . . . . .	9
5.2	Life Cycle Management . . . . .	9
<b>6</b>	<b>Conclusion</b>	<b>9</b>
<b>7</b>	<b>References</b>	<b>10</b>

---

---

## List of Figures

1	XQDDF Overview . . . . .	2
---	--------------------------	---

### Abstract

The standard XQuery language lacks the ability to define and manipulate persistent artifacts like collections, indexes, and integrity constraints. This paper introduces a first attempt to standardize the syntax and semantics of such extensions, and it studies the implications on the static context, dynamic context and processing model of XQuery while dealing with persistent data. The paper presents example modules that show how collections, indexes, and integrity constraints are declared, created, maintained, or accessed.

## 1 Introduction

XQuery has been designed by the World Wide Web Consortium as a general purpose XML information processing language, useful in a variety of architectures and environments. For example, XQuery can be used to process *XML data on the edge* of existing software architectures, where the information is temporary, and is being searched, transformed, or modified, just before being passed along for further processing to other programming languages (e.g. SQL, JAVA, Python, Ruby, JavaScript). Another increasingly popular usage of XQuery is in *XML databases* or *XML end-to-end architectures*. In such architectures, XML is the primary form in which the information is stored and being processed, the information is *persistent* across successive invocations of programs, and XQuery is the primary language for accessing this information for search, filter, transform, update, and for writing more complex application workflows.

Unfortunately, XQuery, as it is currently standardized by the W3C, is incomplete and cannot be used as such (without proprietary language extensions, or rich APIs from other programming languages) in the second type of architectures: persistent databases or XML end-to-end architectures. Unlike its cousin query language, SQL, XQuery lacks the capability to model, describe, and reason about the persistent state of the "database". XQuery 1.1 does indeed have the capability to access collections of nodes at runtime, which could be envisioned as modeling the persistent state of the XML database, yet the language is underspecified in this area. Such collections have no detailed semantics (about copy, order, or multiplicity for example), the language lacks the ability to declare statically such collections, it lacks the static and/or dynamic information that is required for proper compilation and/or execution (e.g. type, update patterns), and it lacks operations to create and modify such collections. Moreover, the language lacks the ability to declare and manage access structures (e.g. indexes) and integrity constraints.

All such concepts are required for a complete XML/XQuery database story. Unless such concepts are included in the standard language itself, each XQuery implementation will have proprietary extensions to overcome such limitations, or such functionalities will be supplied through non XQuery rich APIs. In both cases, the portability of XQuery applications will be limited or the simplicity and elegance of XML end-to-end architectures will be hurt.

In this document, we describe an extension of XQuery 1.1 [[XQUERY11](#)] called XQuery Data Definition Facility (or XQDDF) to deal with such persistent artifacts: collection, indexes, and integrity constraints. The document describes the lifetime and evolution of such artifacts: how are they declared, how do they come into existence, how are they used in the compilation and execution of XQuery programs, and how are they shared by multiple XQuery programs.

Specifically, XQDDF extends

1. the static context with the definitions of collections, indexes, and integrity constraints
2. the dynamic context with the runtime aspects of collections, indexes, and integrity constraints
3. the syntax (and semantics) of the prolog of library modules with the declaration of collections, indexes, and integrity constraints
4. the semantics of the import module statement
5. the XQuery Update Facility [[XQUERYUPDATE10](#)]
  - with new update primitives for creating, deleting, and modifying collections and indexes
  - by adding new expressions for modifying collections

the Function and Operators [[XQUERY10FUNCTIONS](#)] by adding functions for creating, deleting, and modifying collections and indexes, and activating integrity constraints. All such functions are in a new namespace whose prefix in this document is *XQDDF*.

All such extensions are described along the lines of tiny XQuery snippets which are iteratively build up on each other. They illustrate how to declare, manage, and use each of the components of XQDDF. Please note that the purpose of this document is not to be a complete specification. Instead, such a specification is provided online at [[XQDDF10](#)].

The remainder of this paper is describes as follows: First, we give an overview over the different components of XQDDF and their impact on the XQuery processing model, i.e. their impact on the static- and dynamic context. After that, we illustrate collections, indexes, and integrity constraints, each in a separate section. At the end, we conclude and give an outlook in Sec.

## 2 Overview

According to the XQuery 1.1 specification [XQUERY11], collections are sequences of nodes that are potentially available using the `fn:collection` function. Unfortunately, as we mentioned before, XQuery 1.1 collections have no static information, and there underspecified in many aspects. This specification introduces a new kind of collection, together with a complete semantics for declaring, creating, modifying and accessing them. In the remaining of the document by collections we will refer to this new kind of collections introduced in this specification, and not the XQuery 1.1 notion.

XQDDF collections are disjoint sequences of parent-less nodes identified by QNames (not URIs). The sequence of nodes can be retrieved using the `xqddf:collection(QName)` function. They are created by invoking specific functions. Their content can be modified either by specific expressions (e.g. `insert`, `delete`) or by invoking the equivalent side-effecting functions.

Indexes are access structure whose contents are defined by a "domain" expression defining the set of nodes to be indexed and a number of "key" expressions on which the index is being built. Indexes can be either used implicitly by the query processor (if such an evaluation is equivalent to the original program) or used explicitly in expressions using `xqddf:probe` functions. While defining an index there are a lot of questions to be considered and answered: is it a multi-key index or a single-key index? is the index required to have homogeneous set of keys? which kind of equality or comparisons is the index able to solve (value comparisons, general comparisons)? is the index able to solve only equality point search or it is able to solve range queries? how is the index maintained (automatically or explicitly)? is the index maintained up-to-date with respect to the original data in an atomic fashion or can be updated asynchronously? how is the indexed used in programs (automatically used by the compiler or explicitly used by the user)? is the index stable, i.e. does it return the nodes in the (original) order in the collection or not?

This document attempts to give users control over the answer to such questions while they define and manipulate indexes. If the XML data is stored (e.g. relational stores, LDAP stores), the propagation of updates on collections or indexes to such an underlying persistent store is beyond the scope of this specification.

Integrity constraints (ICs) specify rules that ensure the accuracy and consistency of data that is available in collections. One can imagine lots of different kinds of integrity constraints (e.g. uniqueness of keys, foreign keys, or global collection integrity constraints). This document attempts to gives syntax and semantics for a comprehensive and useful set of such constraints, and a way to activate and deactivate them at runtime.

The collections, indexes, and integrity constraints have a dual representation: (a) the static information about such persistent artifacts – that is populated though compilation of their declarations (i.e. during the static analysis phase) – and (b) the dynamic context (runtime) aspect – that is independently populated via the execution of specific creation functions (i.e. during the dynamic analysis phase). The set of statically known collections, indexes, and integrity constraints do not have to be perfectly consistent with the set of dynamically available such artifacts: each dynamic artifacts has to be described in the static context, but the reverse is not required. A runtime access to an artifact that is not found in the dynamic context will result in an execution error.

In general, collections, indices, and ICs are expected to be persistent artifacts, that is, they may survive across and be shared by multiple XQuery programs. This is accomplished by sharing the same XQuery static and dynamic context across programs.

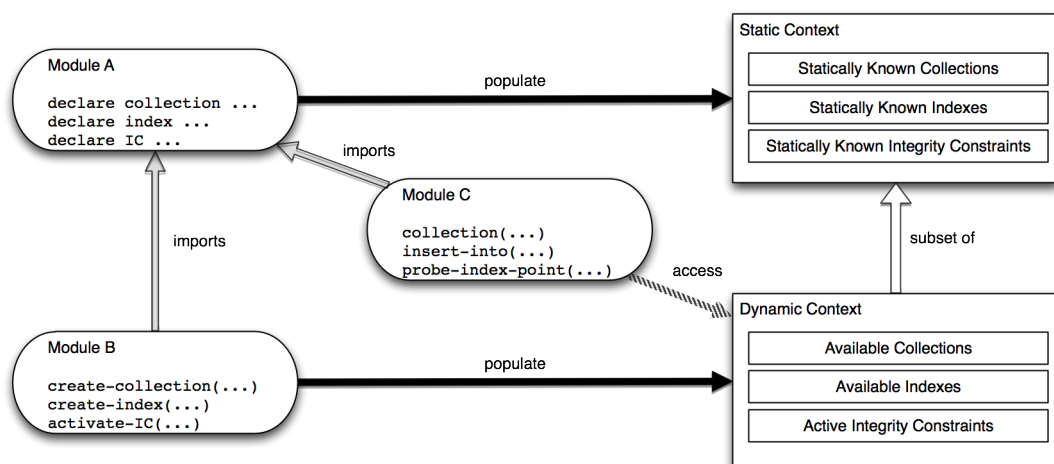


Figure 1: XQDDF Overview

Figure ?? shows an example. In this example, a first module (Module A) declares the collections and indexes and, hence, populates a static context with the information. A second module (Module B) executes in this static context and creates the dynamic structures for collections and indexes, hence modifying/populating the dynamic context. A third module (Module C) executes in the static and dynamic context populated by the first two, and hence can use the persistent artifacts (created by the previous two: collections and indexes).

## 3 Collections

Let us assume an application that models a news organization. The application models its data as XML documents grouped into collections of logically related entities. In the example introduced in this section, we show how three such collections may be created and used; the first collection contains employee data, the second contains news articles, and the third contains information about the months of the year (e.g., the name, number of days, and fixed holidays for each month).

### 3.1 Declaration

Before a collection can be created, it must be declared. A collection declaration is an extension to the prolog of XQuery's library modules and describes the collection by providing (1) a unique name (as QName), (2) specifying certain properties for the collection itself (e.g. modifier properties), and specifying properties for the documents contained in the collection (e.g. types).

In the example query presented below, the declaration of the collections are placed inside the "news-data" library module. These declarations assign the names news-data:employees, news-data:articles, and news-data:months to the three collections, respectively. Documents in both, the employees and the months collections are assumed to have a well-known structure. This structure is reflected in an XML schema ("news-schema") which declares two global elements for employees and months, respectively. Accordingly, the collection declarations for employees and months specify that their (root) nodes are elements whose name and type matches the name and type of the corresponding global element declarations in the "news-schema". In contrast, articles may come from various sources and, as a result, article documents do not have any particular schema. Therefore, the declaration for the articles collection specifies node() as its type. Both employee and article documents may be updated during their lifetime. In contrast, the months-related information is fixed (i.e. it can not change), so the nodes of the months collection are declared as "read-only". Furthermore, the collection itself is declared "const", meaning that no months may be added to or deleted from this collection after it is created and initialized. Finally, we want the order of the month documents within their containing collection to be the same as the actual order of the months within the year. To achieve this, we have to declare the collection as "ordered", so that when we later insert the month documents in the collection, the system will store and return them in the same order as their insertion order. In contrast, the position of employees or articles inside their respective collections does not have any special meaning for the application. Hence, the corresponding declarations do not specify any ordering property. This allows the system to store and access the contents of these collections in what it considers as the optimal order.

```
module namespace news-data = "http://www.news.org/data";

import schema namespace news = "http://www.news.org/schemas";

declare collection news-data:employees
  as schema-element(news:employee)*;

declare collection news-data:articles as node()*;

declare const ordered collection news-data:months
  as schema-element(news:month)*
  with read-only nodes;

declare variable $news-data:employees := xs:QName("news-data:employees");
declare variable $news-data:articles := xs:QName("news-data:articles");
declare variable $news-data:months := xs:QName("news-data:months");
```

Besides declaring collections, the XQuery snippet also declares three variables whose values are the QNames of the collections. Those variables might be used by functions in (other) modules in order to refer to those collections.

### 3.2 Life Cycle Management

Having been declared, the collections can now be created, i.e. added to the dynamic context. Collection creation is illustrated by the XQuery script shown below. First, the collection descriptions must be made visible to the script. This is done by importing the “news-data” module that contains the declarations of the collections. Then, the collections are created by calling the `xqddf:create-collection` function. There exist two variants of this function: the first takes a QName as input (i.e. the `$news-data:employees` variable declared in the “news-data” module) and the second takes both a QName and a node-producing expression. In the first variant, a map is created and registered in the dynamic context that maps collection names to empty sequences. In the second variant, the given expression is evaluated first, and (deep) copies are made of the nodes in the resulting sequence. This way, a sequence of distinct documents is produced which is called the “insertion sequence”. Then, as in the first version of the function, this sequence is added to the dynamic context. In the script below, the latter variant is used to create and initialize the months collection. In fact, months must be initialized during creation because it is a constant collection, so no documents can be added to it later. The months are inserted in the collection in the order from January to December, and since the collection was declared as ordered, this order is preserved in the dynamic context. Collection can be destroyed (i.e. removed from the dynamic context) using the `xqddf:delete-collection` function.

```
import module namespace xqddf = "http://www.zorba-xquery.com/modules/xqddf";

import schema namespace news = "http://www.news.org/schemas";

import module namespace news-data = "http://www.news.org/data";

xqddf:create-collection($news-data:employees);

xqddf:create-collection($news-data:articles);

xqddf:create-collection($news-data:months, (
    validate { <month name="Jan">...</month> },
    ...,
    validate { <month name="Dec">...</month> })
);
```

Note that in the “xqddf” module, most functions are declared as updating, i.e. they return a pending update list (PUL; see *XQUERYUPDATE10*). For example, the first variant of the create-collection functions returns one update primitive to create the collection in the dynamic context. Such changes are only made visible if the corresponding PUL is applied. This can be done (1) either at the end of the query or (2) using an apply expression as provided by the *XQUERYSCRIPTING10*. In the example above, the latter approach is used (see the semi-colon that is used after the function call expression).

### 3.3 Accessing and Deleting Collection Data

In this section, we provide an example that shows how data can be added to collections, deleted from collections, or retrieved from collections. In order to get access to the collections, the corresponding modules and schemas are imported.

Next, the employees collection is populated using the `xqddf:insert-nodes` function. The first argument to this function is the QName of a collection, and the second is a node-producing expression (called the source expression). The QName is used to lookup the collection declaration and the collection itself. Then, the nodes produced by the source expression (source nodes) are copied and the copies are added to the document container, making sure that the actual type of each node matches the static type given in the collection declaration. Copying the source nodes (and their sub-trees) guarantees that the nodes in the insertion sequence (1) are indeed parent-less nodes that do not belong to any other collection already and (2) are distinct from each other. Notice that the need to validate the root nodes against the type specified in the collection declaration is the reason why the “news-schema” must be imported, even though no type defined by the schema is referenced explicitly in the query. In this example, the employees collection is populated by a single call to the `xqddf:insert-nodes` function, whose source expression is a concatenation of explicitly constructed documents.

After populating the collection, the script runs a query expression that uses the `xqddf:collection` function to access the employee and article collections’ root nodes. The expression returns, for each journalist, the articles authored by that journalist ordered by their date.

Finally, the `xqddf:remove-nodes` function is used to remove the articles that were published before 2000 from the article collection. Like `xqddf:insert-nodes`, `xqddf:remove-nodes` takes as input the QName of a collection and a node-producing source

expression. The source nodes must be parent-less nodes that belong to the collection. The function looks up the collection declaration and the collection entry from the dynamic context, and removes the source nodes from the collection. Notice that the whole script is organised as a concatenation of three block expressions (see *XQUERYSCRIPTING10*). Only the second block produces an actual result. The other two are purely updating blocks (their result is the empty sequence and a pending update list). Writing the query as a concatenation of blocks (instead of a single sequential expression), allows the result of the script to be the concatenation of the results of each block (instead of the result of just the last expression in a sequential expression).

```
import module namespace xqddf = "http://www.zorba-xquery.com/modules/xqddf";

import schema namespace news = "http://www.news.org/schemas";

import module namespace news-data = "http://www.news.org/data";

block {
  xqddf:insert-nodes($news-data:employees, (
    validate { <employee id="100">...</employee> },
    ...,
    validate { <employee id="500">...</employee> }
  ));
},
block {
  for $emp in xqddf:collection($news-data:employees)[./position/@kind eq "journalist"]
  let $articles := for $art in xqddf:collection($news-data:articles)[./author//name eq ←
    $emp/news:name]
    order by $art//date
    return $art
  return <result>{$emp}<articles>{$articles//title}</articles></result>
},
block {
  xqddf:delete-nodes($news-data:articles,
    xqddf:collection($news-data:articles)[./date lt xs:date("01/01/2000")]
  );
}
```

## 4 Indexes

Let us assume the same news application that we used in the previous section. In this section, we will illustrate how to create and use indexes on the collections of the news organization. Specifically, we extend this application with two indexes: First, let us assume that each employee has a city where she is currently stationed at. We want to create an index that maps city names to the employees that are stationed in those cities. The first index will contain one entry for each city where at least one employee is stationed in. Moreover, let us assume that we want to search for journalists based on the number of articles they have written. For this, we will create a second index that maps article counts to the employees who are journalists and have produced that number of articles.

### 4.1 Declaration

Similar to collections, indexes must be declared before they can be created. An index declaration describes the index by providing its domain expression, its key expressions, and certain index properties; it also specifies a name for referencing the index in subsequent operations. Like collections, indexes are declared inside the prolog of library modules.

The example below, contains the declaration of two indexes. The first index declaration assigns the name `news-data:CityEmp` to the index. It uses the “on nodes” and “by” keywords to specify the domain and key expressions, respectively. The “as” keyword specifies a target atomic data type which the results of the key expression must match with (after atomization). The index is declared as a “value equality” index. This means that it can be used to find the employees in a particular city, but not in a “range” of cities. In other words, the index is not aware of any ordering among city names. Finally, the maintenance property of the index is set to “automatically maintained”. Briefly, an automatically maintained index is one whose maintenance is the responsibility of the XQuery processor rather than the XQuery programmers.

The second index declaration assigns the name `news-data:ArtCountEmp` to the index. Its domain expression selects all employees who are journalists. Its key expression computes the number of articles written by the “current” journalist. This index is declared as a “value range” index, which means that it can be used to find journalists whose article count is within a given range. Finally, the index is declared as “manually maintained”, which means that programmers must explicitly request the index to be synchronized with the underlying data.

```

module namespace news-data = "http://www.news.org/data";

import schema namespace news = "http://www.news.org/schemas";

import module namespace xqddf = "http://www.zorba-xquery.com/modules/xqddf";

(: collection declarations go here :)

declare automatically maintained value equality index news-data:CityEmp
  on nodes xqddf:collection(xs:QName("news-data:employees"))
  by ../news:station/news:city as xs:string;

declare manually maintained value range index news-data:ArtCountEmp
  on nodes xqddf:collection(xs:QName("news-data:employees"))[../news:position/@kind eq " ←
    journalist"]
  by count(for $art in xqddf:collection(xs:QName("news-data:articles"))
    where $art/empid = ./id
    return $art) as xs:integer;

(: variables for collection QNames go here :)
declare variable $news-data:CityEmp := xs:QName("news-data:CityEmp");
declare variable $news-data:ArtCountEmp := xs:QName("news-data:ArtCountEmp");

```

As for collections, the module declares two variables that may be used by subsequent expressions to access the indexes, e.g. as first parameter to the `xqddf:create-index` function described below.

## 4.2 Life Cycle Management

Having declared the indexes in the “news-data” library module, they can now be created. This is done using the admin-script shown below. This script must first import the “news-data” module. As far as indexes are concerned, the effect of this import is to create two entries in the static context of the main module, mapping the index names to the index definitions (domain expression, key specification, and properties). Then, the query creates the indexes by invoking the `ddf:create-index` function, passing the name of the index as input.

Let us consider the creation of the `CityEmp` index (the process is the same for the `ArtCountEmp` index). Index creation starts with retrieving the index definition from the static context, using the index name. Then, an index container is created, whose entries will be pairs associating a city name with a set of employees. Next, this container is populated using the following process: The domain expression is evaluated, and for each employee node `E` in the domain sequence, the name of the city `C` where the employee is currently stationed in is retrieved by evaluating the key expression, atomizing its result, and checking that the atomic value matches the specified target type. Finally, the pair `[E, C]` is inserted in the index: if an entry for `C` exists already, `E` is inserted in the set associated with `C`; otherwise, a new entry is created mapping `C` to the set `{ E }`. The last step in index creation involves registering the index inside an indexes table in the dynamic context that maps index names to index containers. The index container will remain registered until it is destroyed by a call to the `ddf:delete-index` function.

```

import module namespace xqddf = "http://www.zorba-xquery.com/modules/xqddf";

import module namespace news-data = "http://www.news.org/data";

xqddf:create-index($news-data:CityEmp);

xqddf:create-index($news-data:ArtCountEmp);

```

## 4.3 Probing and Maintaining Indexes

The next step in this example is to show how the index can be used to optimize query performance, which of course, is the primary motivation for supporting indexes in any data-processing system. XQDDF provides two functions for index probing: `probe-index-point` and `probe-index-range`. The latter function is only available for indexes declared as “value range”.

### 4.3.1 Probing Indexes

The first query below illustrates the use of the `xqddf:probe-index-point` function. This query returns the names of all employees stationed in Paris. As shown, the `xqddf:probe-index-point` function takes the index name and the keyword “Paris” as inputs. It uses the index name to find the index container via the indexes tables, looks-up the entry for “Paris” inside this container, and returns all the associated employee nodes.

```
import module namespace xqddf = "http://www.zorba-xquery.com/modules/xqddf";

import module namespace news-data = "http://www.news.org/data";

xqddf:probe-index-point($news-data:CityEmp, "Paris")
```

The second query illustrates index probing via the `xqddf:probe-index-range` function. It returns all journalists who have written more than 100 articles. As shown, the first parameter of the `xqddf:probe-index-range` function is the index name, followed by six parameters per key expression. The six parameters specify a range of values for the key values: the first two are the lower and upper values of the range, the next two are booleans that specify whether the range does indeed have a lower and/or upper bound, and the last two are also booleans that specify whether the range is open or closed from below or above (i.e., whether the lower/upper bound are included in the range or not).

```
import module namespace xqddf = "http://www.zorba-xquery.com/modules/xqddf";

import module namespace news-data = "http://www.news.org/data";

xqddf:probe-index-range($news-data:ArtCountEmp, 100, (), true(), false(), true(), false())
```

### 4.3.2 Maintaining Index Data

Now, let us consider what happens when the data on which an index is built gets updated. In general, index maintenance is the operation where the index contents are updated so that they reflect the index definition with respect to the current snapshot of the data. There exist two maintenance modes: manual and automatic. If an index is declared as “manually maintained”, index maintenance is done only when the updating function `ddf:refresh-index` is invoked inside a query. Essentially, in manual mode maintenance invoking this function is the responsibility of the programmer, and the index may become stale between two consecutive calls to the `ddf:refresh-index` function. In contrast, if an index is declared as “automatically maintained”, the query processor guarantees that the index stays up-to-date at any given time.

Consider the following updating query as an example:

```
import module namespace xqddf = "http://www.zorba-xquery.com/modules/xqddf";

import module namespace news-data = "http://www.news.org/data";

replace node value
  xqddf:collection($news-data:employees)/employee[@id eq "007"]//station/city
with "Beijing"
```

In this example, the `CityEmp` index was declared as automatic. The index-maintenance query transfers the employee with id “007” from his current city, say Paris, to Beijing. Since index `CityEmp` is automatic, after the update is applied, the query processor will initiate a maintenance operation on the index, whereby the employee node will be removed from the node set associated with Paris and inserted into the node set associated with Beijing (if there is no other employee stationed in Beijing already, an entry for it will be created first). Notice that, although the index is not explicitly referenced anywhere in this query,

its definition must still be available to the query because it is needed to perform the index maintenance. In this example, the query imports the “news-data” module because it contains the declaration of the employees collection, which is referenced by the query.

The ArtCountEmp index is more complex than the CityEmp index, so the system may not be able to maintain it in an efficient way. Furthermore, the index contains “statistical” information, so it may be acceptable if its contents are not always in sync with the underlying data. Therefore, the ArtCountEmp index was declared as “manually maintained”.

## 5 Integrity Constraints

Analogously to collections and indexes, XQDDF defines an additional extension to XQuery library modules which allows the declaration of (static) integrity constraints (ICs). Static ICs<sup>1</sup> can be used to ensure that, in every moment in time, all data which is stored in collections is accurate and consistent according to the semantics of an application. As in the relational world, XQDDF defines several types of ICs: Entity, Domain, and Referential ICs. Entity ICs check for the accuracy and consistency of all nodes in a collection. For instance, a special case of the Entity IC is the IC that checks for unique keys among all nodes in a collection. The Domain IC validates that each node in a collection satisfies a given expression. The Referential IC is used to ensure a foreign key relationship between the nodes in two collections.

In this section, we describe how such ICs are declared in a library module and how a particular IC can be (de-)activated. All ICs are described using examples for the news application. Specifically, we declare ICs for the data stored in the news-data:employees and the news-data:articles collections.

### 5.1 Declaration

As for collections and indexes, ICs must be declared before the user can activate them. An IC declaration specifies (1) the name of the IC for being used by function call to (de-)activate it (see next section), (2) the name of the collection(s) whose data should be validated, and (3) the expression(s) that guarantee the accuracy and consistency of the data. Analogously to indexes, ICs are declared inside the prolog of the library module that declares the collection(s) which is/are referenced by the IC.

#### 5.1.1 Entity Integrity

An Entity IC is used to state the uniqueness of a key among all nodes of a collection. For example, the IC (named news-data:UniqueId) in the example below states that the value of the id attribute of each employee is unique among all other nodes in the news-data:employees collection.

```
declare integrity constraint news-data:UniqueId
  on collection news-data:employees
  node $id check unique key $id/@id;
```

The name of the collection is specified after the "on collection" keyword. The path expression following the "check unique key" keyword returns the value to be checked for uniqueness. The result of this path expression must not be empty and is wrapped to return an atomic value. The variable \$id is successively bound to each node of the news-data:employees collection and available in the check expression..

#### 5.1.2 Domain Integrity

The Domain IC allows the user to specify constraints that a particular node in a collection must satisfy. Domain ICs can be used in addition to XML Schema types or if no XML schema is available.

With the following example, we want to make sure that the name of each author of an article is not the zero length string. This can be particularly useful since there is no XML schema for articles.

```
declare integrity constraint news-data:AuthorNames
  on collection news-data:articles
  foreach node $article check fn:string-length($article/author/name) != 0;
```

<sup>1</sup>Note that XQDDF doesn't define any dynamic integrity constraints which check the validity of a particular *update*.

The name of the IC is `news-data:AuthorNames` and it is defined on nodes belonging to the `news-data:articles` collection. The "foreach node" expression specifies a variable (using a QName) which is bound to each node in the collection. For each such node, the check expression is executed. For each node, the boolean effective value of the result of this expression must be equal to true.

### 5.1.3 Referential Integrity

The Referential IC requires every value of a node in a collection to exist as a value of another node in another collection. For example, in the database of the news organization, we want to make sure that each article is maintained by an (existing) employee. This can be done by declaring a so called foreign key IC. In the following example, this IC is given the name `news-data:ArticleEmployees`.

```
declare integrity constraint news-data:ArticleEmployees
  foreign key
    from collection news-data:articles node $x key $x/empid
    to   collection news-data:employees node $y key fn:data($y/@id);
```

The QName following the "from collection" and "to collection" keywords specify the source and destination collections, respectively. Each result of the key expressions are wrapped to return an atomic value. For each atomic value in the source collection, an atomic value in the sequence returned by the key expression on the destination collection must exist. The IC is violated if this is not the case for any node in the source collection. This semantics is equivalent to the following XQuery expression.

```
every $x in xqddf:collection(xs:QName("news-data:articles"))
satisfies
  some $y in xqddf:collection(xs:QName("news-data:employees"))
satisfies $y/id eq $x//sale/empid
```

## 5.2 Life Cycle Management

ICs can be checked manually (if requested by the user) or automatically on updates apply time, after validation and indexes are computed. In order to be checked automatically, an IC needs to be active. ICs can be (de-)activated using the two updating functions `xqddf:activate-integrity-constraint` and `xqddf:deactivate-integrity-constraint`, respectively. Each function takes the name of the IC to (de-)activate as parameter. The flag indicating whether an IC is active or not is stored in the dynamic context.

Deactivating an IC might be useful if the corresponding check is expensive and, hence, inconsistency of the data might be acceptable and only checked (and fixed manually) from time to time. To check an IC manually, the XQDDF defines an updating function called `check-integrity-constraint` which triggers the IC, identified by a QName passed as parameter, to be checked.

Similar to collections and indexes, the module declaring the integrity constraints (i.e. with namespace `http://www.news.org/data`) can also declare variables whose values are the QNames of the ICs. This allows their names to be easily referenced by subsequent expressions. For example, such a variable can be passed as a parameter to the `activate-integrity-constraint` in the importing `admin-script` module (see above). For the ICs from the section above, those variables are declared as follows:

```
declare variable $news-data:UniqueId := xs:QName("news-data:UniqueId");
declare variable $news-data:AuthorName := xs:QName("news-data:AuthorNames");
declare variable $news-data:ArticleEmployees := xs:QName("news-data:ArticleEmployees");
```

## 6 Conclusion

In this document, we have presented an extension to XQuery called the XQuery Data Definition Facility (XQDDF). It extends XQuery with three artifacts: collections, indexes, and integrity constraints. The document describes - using examples - the lifetime and evolution of such artifacts: how they are declared, how they come into existence, how they are used in XQuery programs, and how they are shared among multiple XQuery programs.

In its current state, XQDDF is implemented in version 1.0 of the Zorba XQuery processor ([ZORBA10]) and version 1.0 of Sausalito ([SAUSALITO10]). Sausalito implements an XML end-to-end architecture that allows to create RESTful web applications - entirely in XQuery and deploy them in the cloud. In its heart, Sausalito implements an XML database whose collections, indexes, and integrity constraints are declared and managed using components of XQDDF.

## 7 References

- [XQUERY10] Scott Boag et al. XQuery 1.0: An XML Query Language. W3C Recommendation, 23 January 2007. <http://www.w3.org/TR/2007/REC-xquery-20070123/>
- [XQUERY11] Jonathan Robie et al. XQuery 1.1: An XML Query Language. W3C Working Draft, 15 December 2009. <http://www.w3.org/TR/2009/WD-xquery-11-20091215/>
- [XQUERYUPDATE10] Don Chamberlin et al. XQuery Update Facility 1.0. W3C Candidate Recommendation, 09 June 2009. <http://www.w3.org/TR/2009/CR-xquery-update-10-20090609/>
- [XQUERY10FUNCTIONS] Ashok Malhotra et al. XQuery 1.0 and XPath 2.0 Functions and Operators. W3C Recommendation, 23 January 2007. <http://www.w3.org/TR/2007/REC-xpath-functions-20070123/>
- [XQUERYSCRIPTING10] Don Chamberlin et al. XQuery Scripting Extension 1.0. W3C Working Draft, 3 December 2008. <http://www.w3.org/TR/2008/WD-xquery-sx-10-20081203>
- [XQDDF10] Cezar Andrei et al. XQuery Data Definition Facility 1.0. <http://www.zorba-xquery.com/xqddf10.pdf>
- [ZORBA10] Zorba XQuery Processor 1.0 <http://www.zorba-xquery.com/>
- [SAUSALITO10] Sausalito - An XQuery Application Server for the Cloud <http://www.28msec.com/>
-