

Extending XQuery with collections, indexes and integrity constraints

Cezar Andrei¹, Matthias Brantner², Daniela Florescu¹, David Graf², Donald Kossmann²,
and Markos Zacharioudakis¹

¹Oracle

²28msec Inc.

December 21, 2009

Abstract

The standard XQuery language lacks the ability to define and manipulate persistent artifacts like collections, indexes and integrity constraints. This document 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.

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 the 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 1.0 does indeed have the capability to access at runtime *collections* of nodes, 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.

This document proposes an extension of XQuery called XQuery Data Definition Facility (or XQDDF) to deal with such persistent artifacts: collection, integrity constraints, and integrity constraints. The document defines 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 item the semantics of the import module statement
4. the XQuery Update Facility with new PULs for modifying collections and indexes
5. the XQuery Update Facility by adding new expressions for modifying collections
6. the Function and Operators by adding functions for creating and modifying collections and indexes, and manipulating integrity constraints. all such functions are in a new namespace whose prefix in this document is *xqddf*.

This specification is an extension of XQuery 1.0, XQuery 1.1, XQuery Update Facility and XQuery Scripting Facility.

1.1 Overview

According to the XQuery 1.1 specification, 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 collections, 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 `ddl: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 the which the index is being built. They 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 the `ddl:probe` function. While defining an index there are lot of questions to be considered and answered: is it a multi-key index or 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) ? id 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 – does it it return the nodes in the original order in the collection of not ?

This document attempts to give users control over the answer to such questions while they define and manipulate indexes.

If the XML is stored in other stores (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 kind of integrity constraints (e.g. unicity of keys, foreign

keys, global collection integrity constraints, integrity constraints attached to XML schema types). This document attempts to give syntax and semantics for a comprehensive and useful set of such constraints, and way to activate and deactivate them at runtime.

The collections, indexes and integrity constraints will have a dual representation: (a) the static information about such persistent artifacts - that is populated through compilation of their declarations - and (b) the dynamic context (runtime) aspect - that is independently populated via the execution of specific creation functions. 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 artifact 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.

For example a first module can declare the collections and indexes and hence populates a static context with the information. A second module executes in this static context and creates the dynamic structures for collections and indexes, hence modifying the dynamic context. A third module 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). Such a workflow is depicted in the image below.

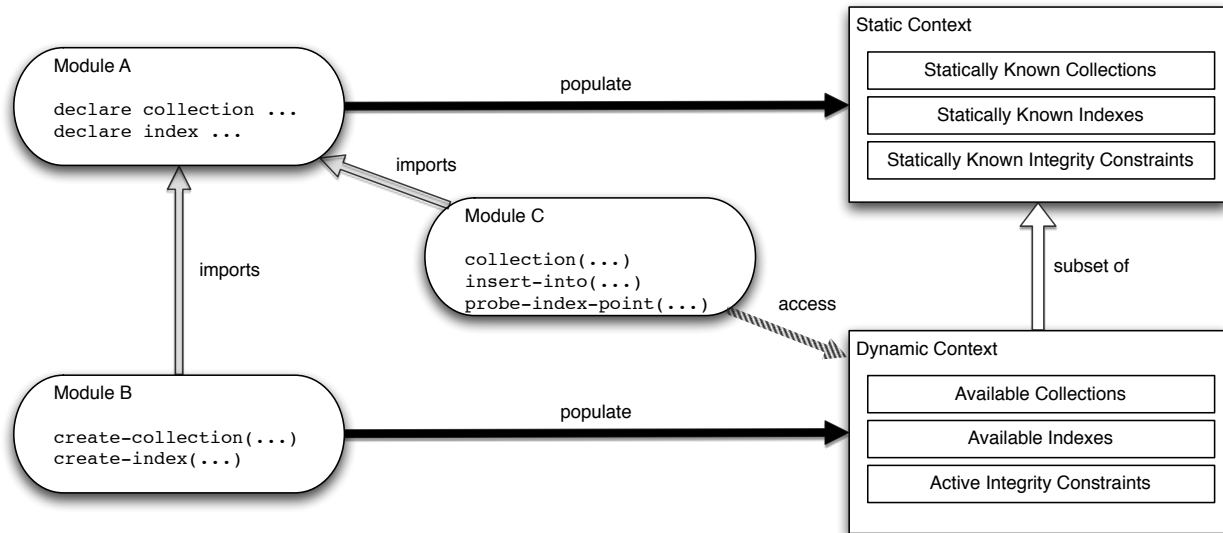


Figure 1: Overview

2 Extensions to the Static and Dynamic Context

The static context contains all the information used for the compilation of XQuery modules (e.g. imported schemas, namespaces, imported modules). The dynamic context contains all the artifacts required for execution (e.g. the actual data). This specification extends both the static and dynamic context with information about collections, indexes and integrity constraints and introduce consistency constraints rules between the static and dynamic context.

2.1 Static Context

The static context is extended with the statically known collections, the statically known indexes and the statically known integrity constraints.

Statically known collections Defines the set of collections that are available via the `xqddf:collection()` function or the collection call expression. Each collection is uniquely identified by its expanded QName. In addition to the name, each statically known collection has a set of properties that controls the behavior of accesses and updates to the collection (see Collection Declaration). Available properties are: the *static type of the collection*, the collection modifier property, and the *node modifier property*. *The static type is the type declared as a static type for the sequence of nodes which represents the content of the collection. The collection modifier property determines which updates are allowed on the collection. Possible values for this property are: read-only, mutable, append-only or queue. The node modifier property determines which updates are allowed on the nodes that are the members of the collection. Possible values for this property are: read-only, mutable.*

Statically known value indexes Defines the set of indexes that are known at static (compile) time. Each index is uniquely identified by its expanded QName. In addition to the name, each statically known index contains an index definition that contains the domain expression and the key expressions, and a set index properties that control the behaviour of accesses and updates to the index. The domain and key expressions are XQuery expressions. *Statically known index properties are: the uniqueness property, the comparison property, and the maintenance property.* The uniqueness property dictates whether an index key tuple can have more than one associated domain nodes. Possible values are unique and non unique. The comparison property determines whether the index keeps its key tuples ordered or not, and hence the index can solve point equality searches, or range searches. Available values are point search and range search. The maintenance property determines whether the maintenance of the index is initiated explicitly by the programmer or done automatically by the system. Available values are automatically or manually maintained. More details about the properties of an index are provided in the description of the Index Declaration.

Statically known integrity constraints Defines the set of declared integrity constraints. Each integrity constraint is uniquely identified by its expanded QName. Each kind of integrity, depending on its type (unique, foreign key, etc) will hold more information about its definition (e.g. the collection(s) it refers to, the key expressions). All statically known integrity constraints are assumed to be satisfied during compilation of the XQuery programs.

2.2 Dynamic Context

The dynamic context is extended with the dynamically available collections, the dynamically available indexes and the dynamically active integrity constraints.

Dynamically available collections This is a mapping of expanded QNames onto sequences of nodes. The associated sequence of nodes is retrieved by the `xqddf:collection` function when this QName is supplied as the argument or using the collection call expression. All the nodes in this sequence as well as all of their descendant nodes are considered as belonging to the collection.

Dynamically available indexes This is a mapping from expanded QNames onto a map containing index entries. Each entry in the map is a [set of key values, domain node] pair. The domain nodes associated with a certain key can be retrieved by the `xqddf:probe-index-point` and `xqddf:probe-index-range` functions (see 6.2.4) when the QName is supplied as first argument to those functions.

Active Integrity Constraints This is a set of QNames, that describes the names of all integrity constraints that are checked at runtime. Integrity constraints are enabled or disabled for run-time check through `xqddf:activate-integrity-constraint` and `xqddf:deactivate-integrity-constraint` functions.

2.3 Consistency Rules between the Static and Dynamic Context

In order to ensure proper execution of XQuery programs, the content of the static and dynamic contexts should satisfy several consistency rules, listed below.

- The set of available collections must be a subset of the set of statically known collections.
- The sequence of nodes that is the content of a dynamically available collection must satisfy (using the semantics of treat-as) the static type declared for this collection as part of the static context.
- The set of available indices must be a subset of the set of statically known indices.
- The dynamic content of an index should satisfy the static definition.
- Each active integrity constraint should be a member of the statically known integrity constraints.
- Each active dynamic constraint is satisfied given the content of the dynamically available collections.

3 Extensions to the Processing Model

The XQuery Data Definition Facility defines the following extension to the XQuery 1.1 processing model.

In the static analysis phase, the static context of an importing module is extended with function declarations, variable declarations, collection declarations, index declarations, and integrity constraint declarations of the imported modules.

4 Extensions to Modules and Prologs

The Prolog of an XQuery module is extended with declarations for collections, indexes, and integrity constraints.

Moreover, the Module Import is extended in order to be able to import collections, indexes, and integrity constraints.

4.1 Prolog

```
Prolog := ( ( DefaultNamespaceDecl | Setter | NamespaceDecl | Import ) Separator)*  
         ( ( VarDecl | ContextItemDecl | FunctionDecl | OptionDecl | CollectionDecl | IndexDecl | ICDecl ) Separator  
         )*
```

A *Prolog* is a series of declarations and imports that define the processing environment for the module that contains the Prolog. Each declaration or import is followed by a semicolon. A Prolog is organized into two parts.

The first part of the Prolog consists of setters, imports, namespace declarations, and default namespace declarations. *Setters* are declarations that set the value of some property that affects query processing, such as construction mode, ordering mode, or default collation. Namespace declarations and default namespace declarations affect the interpretation of QNames within the query. Imports are used to import definitions from schemas and modules. Each *imported schema* or *module* is identified by its target namespace, which is the namespace of the objects (such as elements, functions, collections, indexes, or integrity constraints) that are defined by the schema or module.

The second part of the Prolog consists of declarations of variables, functions, options, collections, indexes, and integrity constraints. These declarations appear at the end of the Prolog because they may be affected by declarations and imports in the first part of the Prolog.

4.2 Collection Declaration

```
CollectionDecl := 'declare' CollProperties 'collection' QName ('as' KindTest OccurrenceIndicator?)? ('with'  
    NodeModifier 'nodes')?  
    ('as' KindTest)?
```

```
CollProperties := ('const' | 'mutable' | 'append-only' | 'queue'  
    | 'ordered' | 'unordered')*
```

```
OccurrenceIndicator := '?' | '*' | '+'
```

```
NodeModifier := ('read-only' | 'mutable')
```

A collection declaration adds the expanded QName, collection properties, a node modifier, and a static type of a collection to the set of statically known collections. If the expanded QName of the collection is equal (as defined by the eq operator) to the name of another collection in the set of statically known collections, a static error is raised err:XQDST001

If no modifier property (i.e. `const`, `append-only`, `queue`, or `mutable`) is specified, then `mutable` is added as modifier property to the set of collection properties. If no order property (i.e. `ordered` or `unordered`) is specified, then `unordered` is added as order property to the set of collection properties.

It is a static error if the declaration contains one of the following inconsistencies:

- two or more modifier properties are specified err:XQDST004.
- two or more order properties are specified err:XQDST005
- `queue` or `append-only` together with `unordered` are specified err:XQDST006

If a collection declaration includes a node modifier, the node modifier is added to the static context as the node modifier of the collection.

If no node modifier is specified, the default is `mutable`. If both, `read-only` and `mutable` are specified, a static error is raised err:XQDST007.

If a collection declaration includes a node type, the type is added to the static context as the type of the collection. If no node type is specified, `document-node(element(*, xs:untyped?))*` is added as the type of the collection.

The following examples depict collection declarations that may be found in a library module describing the data of some sort of commercial organization. It is assumed that the URI of the corresponding library module has been associated with the prefix `org`.

```
declare append-only ordered collection org:transactions  
  node read-only;
```

In this example, the collection contains nodes describing the completed transactions within the organization. Such transactions cannot be modified any further, and as a result, the node modifier of the collection is `read-only`. The collection modifiers `append-only` and `ordered` are used because as they are completed, transactions should be placed at the end of the collection, thus making sure that the ordering of the transactions inside the collection reflects the chronology of their completion. Finally, it is assumed that transactions can come in many different formats, and as a result no XML schema has been defined for transactions. Hence, no specific static data type is associated with this collection.

```
declare collection org:employees  
  as schema-element(org:employee);
```

In this example, the collection contains nodes describing the employees of the organization. New employees may be hired and other employees may quit or be fired. Furthermore, the information for each employee may change in many ways. Hence, no explicit value is given for the collection modifier or the node modifier, defaulting to `mutable` for both. On the other hand, it is assumed that all employees obey the same XML schema, so an explicit node-type is given as a schema test.

```

declare const collection org:months
  node read-only
  as schema-element(org:month);

```

In this example, the collection contains nodes describing the 12 months of the year (e.g. the name and number of days of each month). Neither the number of months nor their descriptions can change. Hence, the collection is declared as `const`, `read-only`, and with a specific static type.

4.3 Index Declaration

```

IndexDecl := 'declare' IndexProperties 'index' QName 'on' nodes QName in IndexDomainExpr
  'by' IndexKeySpec (',' IndexKeySpec)*

```

```

IndexProperties := ('unique' | 'non' 'unique'

```

```

  | 'value range' | 'value equality'

```

```

  | 'automatically maintained' | 'explicitly maintained')*

```

```

IndexDomainExpr := PathExpr

```

```

IndexKeySpec := IndexKeyExpr TypeDeclaration OrderModifier

```

```

IndexKeyExpr := PathExpr

```

```

OrderModifier := ('ascending' | 'descending')? ('empty' ('greatest' | 'least'))? ('collation' UriLiteral)?

```

Syntactically, an index is defined by an index declaration statement, which specifies a unique name for an index as a QName, a domain expression, a number of key specifications, and some additional index properties.

An index declaration adds the index definition to the set of statically known indexes of the module containing the declaration. If the expanded QName of the index is equal (as defined by the eq operator) to the expanded QName of another index in the set of statically known indexes, a static error is raised `err:XQDST002`.

If no uniqueness property (i.e. unique or non unique) is specified, then non unique is added as uniqueness property to the set of index properties. If no kind property (i.e. value range or value equality) is specified, then value equality is added as order property to the set of index properties. If no maintenance property (i.e. automatically or explicitly maintained) is specified, then automatically maintained is added as maintenance property to the set of index properties.

It is a static error if the declaration contains one of the following inconsistencies:

- two or more uniqueness properties are specified `err:XQDST008`
- two or more order properties are specified `err:XQDST009`
- two or more maintenance properties are specified `err:XQDST010`

The domain and key expressions must obey the following semantic restrictions:

- must be deterministic `err:XQDST018`
- must not invoke any input functions other than `dc:collection`. Moreover, the argument to each `xqddf:collection` call must be a constant expression `err:XQDST019`. A *constant expression* is an expression that doesn't access the dynamic context.

The sequence generated by the domain expression is called the *domain sequence*. A node in the domain sequence is called *domain node*. In addition to the semantic restrictions above, the domain expression must obey the following restrictions:

- its context item, context position, and context size are considered undefined, and as a result they must not be referenced err:XQDST017,
- it must generate a sequence of nodes err:XQDDY022.
- it must not contain any duplicate nodes err:XQDDY020.
- Each domain node in the domain sequence must belong to a collection that appears in the available collections of the module that contains the index declaration err:XQDDY027.

An *IndexKeySpec* consist of a key expression, a type declaration, and an order modifier. During the evaluation of the key expression, the variable referring to a domain node is bound and may be accessed in any key expression.

The type declaration of a key expression must specify an atomic type, which is not `xs:untypedAtomic` and which has no occurrence indicator or whose occurrence indicator is `?`. err:XQDST015. Each key expression is wrapped (i.e. serves as `CastableExpr`) in a `TreatExpr` where `SequenceType` is the type specified in the type declaration. That is, during expression evaluation, the result of the key expression must match the given type using the rules of `SequenceType` matching err:XQDTY002. The resulting value is considered to be the result of a key expression and is called a **key item**.

The order modifier affects how different values produced by a key expression for different domain nodes are compared with each other. Its syntax, default values, and semantics are the same as the order modifier that appears in the orderspecs of an order by clause.

The following examples show declarations of indices over the collections declared above.

```
declare unique index org:idx_empid
on nodes $e in
dc: collection(xs:QName("org:employees"))//employee
by $e/id;
```

In this example, an index named `org:idx_empids` is declared to map the id of each employee to the node for that employee. The index is unique, because an id cannot be shared by two or more employees. Ids are not considered to have any particular order, and as a result it does not make sense to ask for, say, the employees whose id is greater than some given value. Therefore, the index is declared as `unordered` (the default).

```
declare ordered index example:idx_empsales
on nodes $e in
dc: collection(xs:QName(org:employees"))//employee
by
count(for $sale in collection(xs:QName("org:transactions"))//sale
  where $sale/empid eq $e/id
  return $sale);
```

In this example, an index named `org:idx_empsales` is declared to map the number of sales performed by an employee to the node for that employee. It is possible that two or more employees have the same number of sales, so the index is non-unique (the default). The index is declared as `ordered` because it may be used to efficiently evaluate expressions that need to retrieve employees whose number of sales is in some given range.

```
declare ordered index org:idx_emp_salary_status
on nodes $e in
dc: collection(xs:QName("org:employees"))//employee
by
$e/salary , $e/marital_status;
```

In this example, an index named `org:idx_emp_salary_status` is declared to map the salary and marital status of an employee to the node for that employee. This index may be used to efficiently evaluate expressions that need to retrieve employees whose salary is in some given range and marital status has a given value.

4.4 Integrity Constraint Declaration

Integrity Constraints are defined on one collection or on two collections as foreign key. Syntax for Integrity constraint declaration is the following:

```
ICDecl := 'declare' 'integrity' 'constraint' QName (ICCollection | ICForeignKey)

ICCollection := 'on' 'collection' QName ( ICCollSequence | ICCollSequenceUnique | ICCollNode )

ICCollSequence := '$' QName 'check' ExprSingle

ICCollSequenceUnique := 'node' '$' QName 'check' 'unique' 'key' PathExpr

ICCollNode := 'foreach' 'node' '$' QName 'check' ExprSingle

ICForeignKey := 'foreign' 'key' ICForeignKeySource ICForeignKeyTarget

ICForeignKeySource := 'from' ICForeignKeyValues

ICForeignKeyTarget := 'to' ICForeignKeyValues

ICForeignKeyValues := 'collection' QName 'node' '$' QName 'key' PathExpr
```

An integrity constraint declaration adds the IC to the set of statically known ICs. The IC-s are considered active i.e. checked at run-time only if they are registered in the dynamic context. Using functions activate-integrity-constraint and deactivate-integrity-constraint IC-s become active in the dynamic context.

If the expanded QName of the declaration is equal (as defined by the eq operator) to the name of another IC in the set of statically known ICs, a static error is raised err:XQDST003. It is a static error if any expression used in an integrity constraint declaration has dependencies to the dynamic context err:XQDST011. It is a static error if an integrity constraint is dependent on non-existing collections (forward references are not allowed)err:XQDST012. It is a static error if the source collection is declared in a different module err:XQDST013. Forward references are not allowed with the exception of functions.

"ExprSingle" must return a valid xs:boolean value, otherwise err:XQDDY018 is raised. The "PathExpr" after "unique key" and "foreign key" will be wrapped to return atomic values.

Integrity constraints must be checked at updates apply time, after validation and indexes are computed. Also the IC checks can be triggered on-demand by calling activate-integrity-constraint function.

The following examples show how integrity constraints are defined.

Example 1

```
declare integrity constraint org:icTransactionsSum
  on collection org:transactions $x check sum($x/sale) gt 1000;
```

In this example an IC is declared on the transactions collection, it checks the sum of sale elements of all the nodes in the collection to be more than 1000.

Equivalent with:

```
sum( dc:collection(xs:QName("org:transactions"))/sale ) gt 1000
```

Example 2

```
declare integrity constraint org:icEmployeesIds
  on collection org:employees node $x check unique key $x/@id;
```

In this example the IC checks that values of id attributes are unique within the collection.

```
let $c := dc:collection(xs:QName("org:employees"))
return
(
  every $i in $c
  satisfies
```

```

    exists ( $i/@id )
  )
  and
  ( functx:are-distinct-values( $c/@id ) )

```

Example 3

```

declare integrity constraint org:icTransactionsSale
  on collection org:transactions foreach node $x check $x/sale gt 0;

```

This IC will check for each node in the collection transactions that the value of sale element is greater than zero.

Equivalent with:

```

every $x in dc:collection(xs:QName("org:transactions")) satisfies $x/sale gt 0

```

Example 4

```

declare integrity constraint org:icEmpSalesForeignKey
  foreign key
  from collection org:transactions node $x key $x//sale/empid
  to collection org:employees node $y key $y/id

```

This IC defines a foreign key check from sale/empid in the transactions collection to the id in employees collection, i.e. for each node \$x in transactions collection (\$x//sale/empid should return one value), a node \$y in collection employees should exist where values of \$x//sale/empid and \$y/id being equal.

Equivalent with:

```

every $x in dc:collection(xs:QName("org:transactions"))
satisfies
  some $y in dc:collection(xs:QName("org:employees"))
  satisfies $y/id eq $x//sale/empid

```

4.5 Module Import

```

ModuleImport := import module (namespace NCName =)? URILiteral (at URILiteral (, URILiteral)*)?

```

[Definition: A module import imports the function declarations, variable declarations, collection declarations, index declarations, and integrity constraint declarations from one or more modules into the function signatures, in-scope variables, statically known collections, statically known indexes, and statically known integrity constraints of the importing module.]

It is a static error if the expanded QName of a collection declared in an imported module is equal (as defined by the eq operator) to the expanded QName of a collection declared in the importing module or in another imported module (even if the declarations are consistent) **err:XDST014**. It is a static error if the expanded QName of an index declared in an imported module is equal (as defined by the eq operator) to the expanded QName of an index declared in the importing module or in another imported module (even if the declarations are consistent) **err:XQDST015**. It is a static error if the expanded QName of an integrity constraint declared in an imported module is equal (as defined by the eq operator) to the expanded QName of an integrity constraint declared in the importing module or in another imported module (even if the declarations are consistent) **err:XQDST016**.

Each module has its own static context. A module import imports only functions, variable declarations, collections, indexes, and integrity constraints; it does not import other objects from the imported modules, such as in-scope schema definitions or statically known namespaces. Module imports are not transitive—that is, importing a module provides access only to function, variable, collection, index, and integrity constraint declarations contained directly in the imported module. For example, if module A imports module B, and module B imports module C, module A does not have access to the functions, variables, collections, indexes, and integrity constraints declared in module C.

It is a static error `err:XQST0036` to import a module if the importing module’s in-scope schema types do not include definitions for the schema type names that appear in the declarations of variables, functions (whether in an argument type or return type), collections, indexes, or integrity constraints that are present in the imported module and are referenced in the importing module.

[Definition: A module M1 directly depends on another module M2 (different from M1) if a variable, function, index, or integrity constraint declared in M1 depends on a variable, function, collection, index, or integrity constraint declared in M2.] It is a static error `err:XQST0093` to import a module M1 if there exists a sequence of modules M1 ... Mi ... M1 such that each module directly depends on the next module in the sequence (informally, if M1 depends on itself through some chain of module dependencies.)

Other than the rules for importing collections, indexes, and integrity constraints, the same rules apply as described in the module import section of the XQuery 1.1 specification.

The following example illustrates a module import:

```
import module namespace customers = "http://example.org/customers";
```

5 Extensions to Existing Expressions

5.1 Function Calls

In order to make calls to functions (which could also be considered as calls to views) and collections look the same, the lookup rule for functions in the static context are extended by this specification. This rule is defined as follows:

1. If the expanded QName and number of arguments in a function call match the name and arity of a function signature in the static context this function is called
2. If the function call has no arguments and the expanded QName is contained in the set of statically known collections, the function

```
ddl:collection($name)
```

where \$name is the QName used in the function call is called

3. Otherwise, a static error `err:XPST0017` is raised.

5.2 Insert

```
InsertExpr := 'insert' ('node' | 'nodes') SourceExpr InsertExprTargetChoice
```

```
InsertExprTargetChoice := (('as' ('first' | 'last'))? ('into' TargetExpr | 'into' 'collection' CollectionName ))
```

```
| 'after' TargetExpr ('into' 'collection' CollectionName) ?
```

```
| 'before' TargetExpr ('into' 'collection' CollectionName) ?
```

```
SourceExpr := ExprSingle
```

```
TargetExpr := ExprSingle
```

```
CollectionName := QName
```

The specification of the Insert expression is identical to the specification of the Insert expression in the XQuery Update Facility 1.0, except the following rule that is added at first position:

- If a CollectionName is specified, the Insert expression is rewritten into the following function calls
 - If "first" is specified, `ddl:insert-nodes-first($CollectionName, $SourceExpr)`

- If "last" is specified, `dll:insert-nodes-last($CollectionName, $SourceExpr)`
- If "after" is specified, `dll:insert-nodes-after($CollectionName, $TargetExpr, $SourceExpr)`
- If "before" is specified, `dll:insert-nodes-before($CollectionName, $TargetExpr, $SourceExpr)`
- Else, `dll:insert-nodes-last($CollectionName, $SourceExpr)`

5.3 Delete

`DeleteExpr := 'delete' ('node' | 'nodes') TargetExpr ('from' 'collection' CollectionName)?`

`TargetExpr := ExprSingle`

`CollectionName := QName`

The specification of the Delete expression is identical to the specification of the Delete expression in the XQuery Update Facility 1.0, except the following rule that is added at first position:

- If a collection is specified, the Delete expression is rewritten into the following function call
 - `dll:delete-nodes($CollectionName, $TargetExpr)`

6 XQDDF Functions

The following functions are in the namespace `http://www.zorba-xquery.com/modules/xqddf`. In the following, this namespace is bound to the prefix `xqddf`.

The following set of functions allow the programmer to modify the set of available collections, and their content. The first parameter of each function is a QName. It specifies the collection, index, or integrity constraint to create, delete, modify, or query.

6.1 Collection Functions

6.1.1 create-collection

`declare updating function xqddf:create-collection($name as xs:QName)`

The `create-collection` function is an updating function that adds a mapping from the expanded QName `$name` to an empty sequence to the map of available collections.

Example

```
xqddf:create-collection(xs:QName("example:customers"))
```

The semantics of a `create-collection` function are as follows:

- If the expanded QName of `$name` is not equal (as defined by the eq operator) to the name of any resource in the statically known collections, an error is raised `err:XQDST001`.
- If available collections already provides a mapping for the expanded QName `$name`, an error is raised `err:XQDDY009`.
- The result of the function is an empty XDM instance and a pending update list that consists of the following update primitive:
 - `upd:createCollection($name)` (see 7.1.1)

6.1.2 create-collection (with initial data)

declare updating **function** xqddf:create-collection(\$name as xs:QName, \$content as node(*)*)

The **create-collection** function is an updating function that adds a new mapping from the expanded QName \$name to the map of available collections. Moreover, it adds copies of the sequence \$content to this mapping.

The semantics of a **create-collection** function with initial data are as follows:

- If the expanded QName of \$name is not equal (as defined by the eq operator) to the name of any resource in the statically known collections, an error is raised err:XQDST001.
- If available collections already provides a mapping for the expanded QName \$name, an error is raised err:XQDDY009.
- If \$content does not match the expected type according to the rules for SequenceType Matching, a type error is raised err:XQDTY001.
- The variable \$content is evaluated as though it were an enclosed expression in an element constructor (see Rule 1e in Section 3.7.1.3 ContentXQ of the XQuery Update Facility 1.0 specification). The result of this step is a sequence of nodes to be inserted, called the insertion sequence. Let \$list be the insertion sequence.
- The result of the function is an empty XDM instance and a pending update list that consists of the following update primitives:
 - upd:createCollection(\$name) (see 7.1.1)
 - If \$list is not empty, upd:insertNodesFirst(\$name, \$list) (see 7.1.4)

Example

```
xqddf:create-collection(xs:QName("example:customers"),
    <customer>
      <firstname>Alan</firstname>
      <lastname>Turing</lastname>
    </customer>)
```

6.1.3 delete-collection

declare updating **function** xqddf:delete-collection(\$name as xs:QName)

The **delete-collection** function is an updating function that deletes the mapping with QName \$name from the map of available collections.

Example

```
xqddf:delete-collection(xs:QName("example:customers"))
```

The semantics of a **delete-collection** function are as follows:

- If available collections does not provide a mapping for the expanded QName \$name, an error is raised err:XQDDY009.
- If any of the in-scope variables references a node that belongs to the collection with QName \$name, an error is raised err:XQDDY014.
- If the domain or key expression of any of the available indexes access the collection with name \$name, an error is raised err:XQDDY015.
- The result of the function is an empty XDM instance and a pending update list that consists of the following update primitive:
 - upd:deleteCollection(\$name) (see 7.1.2).

6.1.4 insert-nodes

```
declare updating function xqddf:insert-nodes($name      as xs:QName,  
                                           $content as node()*)
```

The `insert-nodes` function is an updating function that inserts copies of zero or more nodes into a collection.

Remarks Note that the position of the nodes to be inserted is not defined. Moreover, the order of the nodes as they appear in the sequence `$content` after inserted into the collection is not defined.

Example

```
xqddf:insert-nodes(xs:QName("example:customers"),  
                 <customer>  
                   <firstname>Alan</firstname>  
                   <lastname>Turing</lastname>  
                 </customer>)
```

The semantics of a `insert-nodes` function are as follows:

- If available collections does not provide a mapping for the expanded QName `$name`, an error is raised `err:XQDDY006`.
- If the modifier property of the collection `$name` in the set of statically know collections, is `append-only`, `const`, or `queue`, an error is raised `err:XQDDY001`.
- If `$content` does not match the expected type according to the rules for SequenceType Matching, a type error is raised `err:XQDTY001`.
- The variable `$content` is evaluated as though it were an enclosed expression in an element constructor (see Rule 1e in Section 3.7.1.3 ContentXQ of the XQuery Update Facility 1.0 specification). The result of this step is a sequence of nodes to be inserted, called the insertion sequence. Let `$list` be the insertion sequence.
- The result of the function is an empty XDM instance and a pending update list that consists of the following update primitive:
 - If `$list` is not empty, `upd:insertNodes($name,$list)` (see 7.1.3)

6.1.5 insert-nodes-first

```
declare updating function xqddf:insert-nodes-first($name      as xs:QName,  
                                                  $content as node()*)
```

The `insert-nodes-first` function is an updating function that inserts copies of zero or more nodes as first nodes into a collection.

Remarks

- If multiple nodes are inserted by a single function, the nodes remain adjacent and their order preserves the node ordering of `$content`.
- If multiple groups of nodes are inserted by multiple insert functions in the same snapshot, adjacency and ordering of nodes within each group is preserved but ordering among the groups is implementation-dependent.

Example

```
xqddf:insert-nodes-first(xs:QName("example:customers"),  
                       <customer>  
                         <firstname>Alan</firstname>  
                         <lastname>Turing</lastname>  
                       </customer>)
```

The semantics of a `insert-nodes-first` function are as follows:

- If available collections does not provide a mapping for the expanded QName `$name`, an error is raised `err:XQDDY006`.
- If the modifier property of the collection `$name` in the set of statically know collections, is `append-only`, `const`, or `queue`, an error is raised `err:XQDDY001`.
- If the order property of the statically known collection `$name` is `unordered`, an error is raised `err:XQDDY001`.
- If `$content` does not match the expected type according to the rules for SequenceType Matching, a type error is raised `err:XQDTY001`.
- The variable `$content` is evaluated as though it were an enclosed expression in an element constructor (see Rule 1e in Section 3.7.1.3 ContentXQ of the XQuery Update Facility 1.0 specification). The result of this step is a sequence of nodes to be inserted, called the insertion sequence. Let `$list` be the insertion sequence.
- The result of the function is an empty XDM instance and a pending update list that consists of the following update primitive:
 - If `$list` is not empty, `upd:insertNodesFirst($name,$list)` (see 7.1.4)

6.1.6 `insert-nodes-last`

```
declare updating function xqddf:insert-nodes-last($name as xs:QName,  
                                                $content as node(*)
```

The `insert-nodes-last` function is an updating function that inserts copies of zero or more nodes as last nodes into a collection.

The remarks of `insert-nodes-last` are identical to the remarks of `insert-nodes-first`.

Example

```
xqddf:insert-nodes-last(xs:QName("example:customers"),  
                       <customer>  
                         <firstname>Alan</firstname>  
                         <lastname>Turing</lastname>  
                       </customer>)
```

The semantics of a `insert-nodes-last` function are as follows:

- If available collections does not provide a mapping for the expanded QName `$name`, an error is raised `err:XQDDY006`.
- If the modifier property of the collection `$name` in the set of statically know collections, is `const` an error is raised `err:XQDDY001`.
- If the order property of the statically known collection `$name` is `unordered`, an error is raised `err:XQDDY001`.
- If `$content` does not match the expected type according to the rules for SequenceType Matching, a type error is raised `err:XQDTY001`.
- The variable `$content` is evaluated as though it were an enclosed expression in an element constructor (see Rule 1e in Section 3.7.1.3 ContentXQ of the XQuery Update Facility 1.0 specification). The result of this step is a sequence of nodes to be inserted, called the insertion sequence. Let `$list` be the insertion sequence.
- The result of the function is an empty XDM instance and a pending update list that consists of the following update primitive:
 - If `$list` is not empty, `upd:insertNodesLast($name, $list)` (see 7.1.5).

6.1.7 insert-nodes-before

```
declare updating function xqddf:insert-nodes-before($name    as xs:QName,  
                                                $target  as node() ,  
                                                $content as node()*)
```

The `insert-nodes-before` function is an updating function that inserts copies of zero or more nodes as preceding siblings of `$target` into a collection.

The remarks of `insert-nodes-before` are identical to the remarks of `insert-nodes-first`.

Example

```
xqddf:insert-nodes-before(xs:QName("example:customers"),  
                          xqddf:collection($name)[42],  
                          <customer>  
                            <firstname>Alan</firstname>  
                            <lastname>Turing</lastname>  
                          </customer>)
```

The semantics of a `insert-nodes-before` function are as follows:

- If available collections does not provide a mapping for the expanded QName `$name`, an error is raised err:XQDDY006.
- If the modifier property of the collection `$name` in the set of statically know collections, is `append-only`, `const`, or `queue`, an error is raised err:XQDDY001.
- If the order property of the statically known collection `$name` is `unordered`, an error is raised err:XQDDY001.
- If `$content` does not match the expected type according to the rules for SequenceType Matching, a type error is raised err:XQDTY001.
- The variable `$content` is evaluated as though it were an enclosed expression in an element constructor (see Rule 1e in Section 3.7.1.3 ContentXQ of the XQuery Update Facility 1.0 specification). The result of this step is a sequence of nodes to be inserted, called the insertion sequence. Let `$list` be the insertion sequence.
- `$target` is evaluated and checked as follows:
 - If the result is not a node that is contained in the sequence identified by `$name` in the available collections, err:XQDDY006 is raised.
 - Let `$tnode` be the node returned by `$target`.
- The result of the function is an empty XDM instance and a pending update list that consists of the following update primitive:
 - If `$list` is not empty, `upd:insertNodesBefore($name,$tnode,$list)` (see 7.1.6).

6.1.8 insert-nodes-after

```
declare updating function xqddf:insert-nodes-after($name    as xs:QName,  
                                                $target  as node() ,  
                                                $content as node()*)
```

The `insert-nodes-after` function is an updating function that inserts copies of zero or more nodes as following siblings of `$target` into a collection.

The remarks of `insert-nodes-after` are identical to the remarks of `insert-nodes-first`.

Example

```
xqddf:insert-nodes-after(xs:QName("example:customers"),
                        xqddf:collection($name)[42],
                        <customer>
                          <firstname>Alan</firstname>
                          <lastname>Turing</lastname>
                        </customer>)
```

The semantics of `insert-nodes-last` are identical to the semantics of `insert-nodes-before`, except that the last rule is changed as follows:

- If `$list` is not empty, `upd:insertNodesAfter($name, $tnode, $list)` (see 7.1.7).

6.1.9 delete-nodes

```
declare updating function xqddf:delete-nodes($name as xs:QName,
                                             $target as node()+)
```

The `delete-nodes` function is an updating function that deletes zero or more nodes from a collection.

Example

```
xqddf:delete-nodes(xs:QName("example:customers"),
                  xqddf:collection($name)[42])
```

The semantics of a `delete-nodes` function are as follows:

- If available collections does not provide a mapping for the expanded QName `$name`, an error is raised `err:XQDDY006`.
- If the modifier property of the statically known collection `$name` is `append-only`, `const`, or `queue`, an error is raised `err:XQDDY001`.
- `$target` is evaluated and checked as follows:
 - Let `$tlist` be the nodes returned by `$target`.
 - If any nodes in `$tlist` is not part of the sequence that is mapped from the expanded QName `$name`, an error is raised `err:XQDDY029`.
 - If any of the in-scope variables references a node that belongs to `$tlist`, an error is raised `err:XQDDY015`.
- For each node `$tnode` in `$tlist`, the following update primitive is appended to the pending update list: `upd:deleteNode($name$, $tnode)` (see 7.1.8). The resulting pending update list (together with an empty XDM instance) is the result of the `delete-nodes` function.

6.1.10 delete-node-first

```
declare updating function xqddf:delete-node-first($name as xs:QName)
```

The `delete-node-first` function is an updating function that deletes the first node from a collection.

Example

```
let $name := xs:QName("example:customers")
return
  xqddf:delete-node-first($name)
```

The semantics of a `delete-node-first` function are as follows:

- If available collections does not provide a mapping for the expanded QName `$name`, an error is raised `err:XQDDY006`.
- If the modifier property of the statically known collection `$name` is `append-only` or `const`, an error is raised `err:XQDDY001`.

- If the order property of the statically known collection \$name is **unordered**, an error is raised err:XQDDY001.
- If the sequences mapped by the expanded QName \$name doesn't contain any nodes, an error is raised err:XQDDY012.
- Let \$node be the first node of the sequence in the available collections identified by QName \$name.
- If any of the in-scope variables references \$node, an error is raised err:XQDDY014.
- The result of this function is an empty XDM instance and a pending update list containing the upd:deleteNode(\$name, \$node) update primitive (see upd:deleteNode).

6.1.11 delete-nodes-first

declare updating **function** xqddf:delete-nodes-first(\$name as xs:QName, \$number as xs:positiveInteger)

The **delete-nodes-first** function is an updating function that deletes an arbitrary number of nodes from the beginning of a collection.

Example

```
let $name := xs:QName("example:customers")
return
  xqddf:delete-nodes-first($name, 10)
```

The semantics of a **delete-nodes-first** function are as follows:

- If available collections does not provide a mapping for the expanded QName \$name, an error is raised err:XQDDY006.
- If the modifier property of the statically known collection \$name is **append-only** or **const**, an error is raised err:XQDDY001.
- If the order property of the statically known collection \$name is **unordered**, an error is raised err:XQDDY002.
- If the sequences mapped by the expanded QName \$name doesn't contain any nodes, an error is raised err:XQDDY012.
- Let \$nodes be the sequence of the first \$number of nodes of the sequence in the available collections identified by QName \$name.
- If any of the in-scope variables references any node in \$nodes, an error is raised err:XQDDY014.
- Let \$node be a node in the sequence \$nodes. The result of this function is an empty XDM instance and a pending update list containing the update primitive upd:deleteNode(\$name, \$node) update primitive (see upd:deleteNode).

6.1.12 delete-node-last

declare updating **function** xqddf:delete-node-last(\$name as xs:QName)

The **delete-node-last** function is an updating function that deletes the last node from a collection.

Example

```
let $name := xs:QName("example:customers")
return
  xqddf:delete-node-last($name)
```

The semantics of a **delete-node-last** function are as follows:

- If available collections does not provide a mapping for the expanded QName \$name, an error is raised err:XQDDY006.

- If the modifier property of the collection \$name in the set of statically known collections, is `append-only`, `const`, or `queue`, an error is raised `err:XQDDY001`.
- If the order property of the statically known collection \$name is `unordered`, an error is raised `err:XQDDY002`.
- If the sequences mapped by the expanded QName \$name doesn't contain any nodes, an error is raised `err:XQDDY012`.
- Let \$tnode be the last node of the sequence in the available collections identified by QName \$name.
- If any of the in-scope variables references \$tnode, an error is raised `err:XQDDY014`.
- The result of this function is an empty XDM instance and a pending update list containing the `upd:deleteNode($name, $tnode)` update primitive (see `upd:deleteNode`).

6.1.13 delete-nodes-last

```
declare updating function xqddf:delete-nodes-last($name as xs:QName, $number as xs:positiveInteger)
```

The `delete-nodes-last` function is an updating function that deletes an arbitrary number of nodes from the end of a collection.

Example

```
let $name := xs:QName("example:customers")
return
  xqddf:delete-nodes-last($name, 5)
```

The semantics of a `delete-nodes-last` function are as follows:

- If available collections does not provide a mapping for the expanded QName \$name, an error is raised `err:XQDDY006`.
- If the modifier property of the statically known collection \$name is `append-only` or `const`, an error is raised `err:XQDDY001`.
- If the order property of the statically known collection \$name is `unordered`, an error is raised `err:XQDDY002`.
- If the sequences mapped by the expanded QName \$name doesn't contain any nodes, an error is raised `err:XQDDY012`.
- Let \$tnodes be the sequence of the last \$number of nodes of the sequence in the available collections identified by QName \$name.
- If any of the in-scope variables references any node in \$tnodes, an error is raised `err:XQDDY014`.
- Let \$tnode be a node in the sequence \$tnodes. The result of this function is an empty XDM instance and a pending update list containing the update primitive `upd:deleteNode($name, $tnode)` update primitive (see `upd:deleteNode`).

6.1.14 collection

```
declare function xqddf:collection($name as xs:QName)
```

The `collection` function is simple function that returns the sequence of nodes belonging to the entry with QName \$name in the map of available collections.

Example

```
let $name := xs:QName("example:customers")
return
  xqddf:collection($name)
```

The semantics of a `collection` function are as follows:

- If available collections does not provide a mapping for the expanded QName `$name`, an error is raised `err:XQDDY006`.
- If the order property of the statically known collection `$name` is `ordered`, the result of this function is the sequence of nodes from the set of available collections that belongs to the entry with expanded QName `$name`.
- If the order property of the statically known collection `$name` is `unordered`, the result of this function is the sequence of nodes from the set of available collections that belongs to the entry with expanded QName `$name`. The ordering of those nodes is not defined. Note, however, that the document order of each returned node is preserved.

6.1.15 index-of

```
declare function xqddf:index-of($name as xs:QName
                               $node as node()) as xs:integer
```

The `index-of` function is simple function that returns the index of the node `$node` in the sequence of nodes belonging to the entry with QName `$name` in the map of available collections.

Example

```
let $name := xs:QName("example:customers")
let $node := xqddf:collection($name)[42]
return
  xqddf:index-of($name, $node)
```

The semantics of a `index-of` function are as follows:

- If available collections does not provide a mapping for the expanded QName `$name`, an error is raised `err:XQDDY006`.
- If the order property of the statically known collection `$name` is `unordered`, an error is raised `err:XQDDY002`.
- If the node `$node` is not contained in the collection `$name`, an error is raised `err:XQDDY012`.
- The result of this function is the position as `xs:integer` of the node `$node` in the sequence belonging to the entry with QName `$name` in the map of available collections.

6.2 Index Functions

6.2.1 create-index

```
declare updating function xqddf:create-index($name as xs:QName)
```

The `create-index` function is an updating function that adds a mapping from the expanded QName `$name` to a map of index entries to the map of available indexes.

Example

```
xqddf:create-index(xs:QName("example:idx_empsales"))
```

The semantics of a `create-index` function are as follows:

- If the expanded QName of `$name` is not equal (as defined by the `eq` operator) to the name of any resource in the statically known indexes, an error is raised `err:XQDDY004`.
- If available indexes already provides a mapping for the expanded QName `$name`, an error is raised `err:XQDDY010`.

- The result of the function is an empty XDM instance and a pending update list that consists of the following update primitive:
 - `upd:createIndex($name)` (see `upd:createIndex`)

6.2.2 delete-index

declare updating **function** `xqddf:delete-index($name as xs:QName)`

The `delete-index` function is an updating function that removes a resource from the map of available indexes. The QName `$name` is the name of the resource.

Example

```
xqddf:delete-index(xs:QName("example:idx_empsales"))
```

The semantics of a `delete-index` function are as follows:

- If available indexes does not provide a mapping for the expanded QName `$name`, an error is raised `err:XQDDY007`.
- The result of the function is an empty XDM instance and a pending update list that consists of the following update primitive:
 - `upd:deleteIndex($name)` (see 7.1.10).

6.2.3 refresh-index

declare updating **function** `xqddf:refresh-index($name as xs:QName)`

The `refresh-index` function is an updating function that populates the map of index entries identified by the expanded QName `$name` in the map of available indexes.

Note that if the maintenance property of the index in question is automatically maintained, this function is a NOP.

The semantics of a `refresh-index` function are as follows:

- If available indexes does not provide a mapping for the expanded QName `$name`, an error is raised `err:XQDDY007`.
- The result of the function is an empty XDM instance and a pending update list that consists of the following update primitive.
 - `upd:refreshIndex($name)` (see 7.1.11).

6.2.4 probe-index

Probing an index means retrieving the domain items associated with a particular search condition, which is either a single key tuple, or a “range” of key tuples. Probing is done via two functions: `probe-index-point` and `probe-index-range`. Both functions accept a variable number of arguments. The first argument is always a QName identifying an index. The rest of the arguments specify the search condition. For both functions, the index QName must exist in both the statically known indices and the available indices.

probe-index-point For the `probe-index-point` function, the search condition is specified as a number of atomic (or empty) items comprising a key tuple. This number must be equal to the number of `indexspecs` found in the definition of the given index (`err:TDB`). If the index contains an entry with the given key tuple, the associated domain nodes are returned. Otherwise, the empty sequence is returned.

```
probe-index-point($indexQName as xs:QName,
                 $key1      as xs:anyAtomic?,
                 ...,
                 $keyM      as xs:anyAtomic?) as item()*
```

probe-index-range The probe-index-range function can be invoked on ordered indices only (err:TBD). In this case, the search condition is specified as a number of rangespecs.

```

probe-index-range ($indexQName           as xs:QName,
                  $rangeLowerBound1      as xs:anyAtomic?,
                  $rangeUpperBound1      as xs:anyAtomic?,
                  $rangeHaveLowerBound1  as xs:boolean?,
                  $rangeHaveupperBound1  as xs:boolean?,
                  $rangeLowerBoundIncluded1 as xs:boolean?,
                  $rangeupperBoundIncluded1 as xs:boolean?,
                  . . . . ,
                  $rangeLowerBoundM     as xs:anyAtomic?,
                  $rangeUpperBoundM     as xs:anyAtomic?,
                  $rangeHaveLowerBoundM  as xs:boolean?,
                  $rangeHaveupperBoundM  as xs:boolean?,
                  $rangeLowerBoundIncludedM as xs:boolean?,
                  $rangeupperBoundIncludedM as xs:boolean?) as item()*

```

The number of rangespecs must be less or equal to the number of indexespecs found in the definition of the given index (err:TDB). Each rangespec describes a constraint on the values of a key column; the first rangespec applies to the first key column, the second rangespec to the second key column, etc. Each rangespec consists of 6 values:

rangeLowerBound The lower bound in a range of key values.

rangeUpperBound The upper bound in a range of key values.

rangeHaveLowerBound If false, then there is no lower bound, or equivalently, the lower bound is -INFINITY (the actual rangeLowerBound value is ignored). Otherwise, the lower bound is the one given by the rangeLowerBound value. The effective lower bound of the range is either the rangeLowerBound if rangeHaveLowerBound is true, or -INFINITY if rangeHaveLowerBound is false.

rangeHaveUpperBound If false, then there is no upper bound, or equivalently, the upper bound is +INFINITY (the actual rangeUpperBound value is ignored). Otherwise, the upper bound is the one given by the rangeUpperBound value. The effective upper bound of the range is either the rangeUpperBound if rangeHaveUpperBound is true, or +INFINITY if rangeHaveUpperBound is false.

rangeLowerBoundIncluded If false, then the range is open from below, i.e., the rangeLowerBound value is not considered part of the range. Otherwise, the range is closed from below, i.e., the rangeLowerBound value is part of the range.

rangeUpperBoundIncluded If false, then the range is open from above, i.e., the rangeUpperBound value is not considered part of the range. Otherwise, the range is closed from above, i.e., the rangeUpperBound value is part of the range. If the number of rangespecs is less than the number of key columns, then the missing rangespecs are assumed to have the following value: [(, (, false, false, false, false].

A key tuple $K = [k_1, \dots, k_M]$ satisfies a range search condition if for each $i = 1, 2, \dots, M$ the following is true:

$$\text{effectiveLowerBound}_i \text{ lowerCompOp } k_i \text{ upperCompOp}_i \text{ effectiveUpperBound}_i.$$

where lowerCompOp (upperCompOp) is either the le operator if rangeLowerBoundIncluded_{*i*} (rangeUpperBoundIncluded_{*i*}) is true, or the lt operator if rangeLowerBoundIncluded_{*i*} (rangeUpperBoundIncluded_{*i*}) is false. The index-probe-range function finds all key tuples in the index that satisfy the given search condition and returns the the domain items associated with each such key tuple.

6.3 Integrity Constraint Functions

6.3.1 activate-integrity-constraint

```
declare updating function xqddf:activate-integrity-constraint($icQName as xs:QName)
```

The function adds an entry to the dynamically available integrity-constraints. The integrity-constraint QName passed as an argument to the function must exist in the statically known integrity-constraints, otherwise an error is raised `err:XQDDY005`. If an integrity-constraint with the same QName exists already, an error is raised `err:XQDDY011`. Once activated, the IC will automatically be checked by run-time.

6.3.2 deactivate-integrity-constraint

```
declare updating function xqddf:deactivate-integrity-constraint($icQName as xs:QName)
```

The function destroys the integrity-constraint container (if it has been created before), and removes the integrity-constraint from the dynamic context. If deactivated the IC checks will not be enforced by run-time. The QName passed as an argument to the function must exist in the statically known integrity-constraints `err:XQDDY005`.

6.3.3 check-integrity-constraint

```
declare function xqddf:check-integrity-constraint($icQName as xs:QName) as xs:boolean
```

The function triggers checking the IC parameter. It returns true if the defined conditions are met, otherwise false. Error is thrown if during execution any errors are raised.

7 Extensions to Update Operations

7.1 Update Primitives

7.1.1 upd:createCollection

Parameters

```
upd:createCollection(  
  $name as xs:QName  
)
```

Summary Adds a mapping from the expanded QName \$name to an empty-sequence to the map of available collections.

Constraints The expanded QName \$name must be a name of a resource in the statically known collections. Available collections must not provide a mapping for the expanded QName \$name.

Semantics Inserts the mapping from QName \$name to an empty-sequence into available collections.

7.1.2 upd:deleteCollection

Parameters

```
upd:deleteCollection(  
  $name as xs:QName  
)
```

Summary Deletes the mapping for the expanded QName \$name from the set of available collections.

Constraints Available collections must provide a mapping for the expanded QName \$name.

Semantics Deletes the mapping for QName \$name from the map of available collections.

7.1.3 upd:insertNodes

Parameters

```
upd:insertNodes(  
  \ $name as xs:QName,  
  \ $content as node()+  
)
```

Summary Inserts \$content into the collection \$name.

Constraints

- The modifier property of the collection \$name is **mutable**.
- Available collections must provide a mapping from the expanded QName \$name.

Semantics

1. Gets the sequence from the available collections that is mapped from the expanded QName \$name.
2. The sequence is changed to add the nodes in \$content.

7.1.4 upd:insertNodesFirst

Parameters

```
upd:insertNodesFirst(  
  \ $name as xs:QName,  
  \ $content as node()+  
)
```

Summary Inserts \$content as the first nodes into the collection \$name.

Constraints

- The modifier property of the collection \$name is **mutable**.
- The order property of the collection \$name is **ordered**.
- Available collections must provide a mapping from the expanded QName \$name.

Semantics

1. Gets the sequence from the available collections that is mapped from the expanded QName \$name.
2. The sequence is changed to add the nodes in \$content as the first nodes, preserving their order among the nodes in \$content.

7.1.5 upd:insertNodesLast

Parameters

```
upd:insertNodesLast(  
  \ $name as xs:QName,  
  \ $content as node()+  
)
```

Summary Inserts \$content as the last nodes into the collection \$name.

Constraints

- The modifier property of the collection \$name is **mutable append-only**, or **queue**.
- The order property of the collection \$name is **ordered**.

- Available collections must provide a mapping from the expanded QName \$name.

Semantics

1. Gets the sequence from the available collections that is mapped from the expanded QName \$name.
2. The sequence is changed to add the nodes in \$content as the last nodes, preserving their order.

7.1.6 upd:insertNodesBefore

Parameters

```
upd:insertNodesBefore(
  $name as xs:QName,
  $target as node(),
  $content as node()+
)
```

Summary Inserts \$content immediately before \$target.

Constraints

- The modifier property of the collection \$name is **mutable**.
- The order property of the collection \$name is **ordered**.
- Available collections must provide a mapping from the expanded QName \$name.
- \$target must belong to the mapping.

Semantics

1. Gets the sequence from the available collections that is mapped from the expanded QName \$name.
2. The sequence is modified to add the nodes in \$content just before \$target, preserving their order.

7.1.7 upd:insertNodesAfter

Parameters

```
upd:insertNodesAfter(
  $name as xs:QName,
  $target as node(),
  $content as node()+
)
```

Summary Inserts \$content immediately after \$target.

Constraints

- The modifier property of the collection \$name is **mutable**.
- The order property of the collection \$name is **ordered**.
- Available collections must provide a mapping from the expanded QName \$name.
- \$target must belong to the mapping.

Semantics

1. Gets the sequence from the available collections that is mapped from the expanded QName \$name.
2. The sequence is modified to add the nodes in \$content just after \$target, preserving their order.

7.1.8 upd:deleteNode

Parameters

```
upd:deleteNode(  
  $name as xs:QName,  
  $target as node()  
)
```

Summary Deletes \$target from collection \$name.

Constraints Available collections must provide a mapping from the expanded QName \$name.

Semantics

1. Gets the sequence from the available collections that is mapped from the expanded QName \$name.
2. If the node \$target is part of the sequence, it is deleted from the sequence.

Notes

- Removed nodes are detached from their collections; however, a node removal has no effect on variable bindings.
- Multiple upd:deleteNode operations may be applied to the same node during execution of a query; this is not an error.

7.1.9 upd:createIndex

Parameters

```
upd:createIndex(  
  $name as xs:QName  
)
```

Summary Adds an entry to the map of available indexes and populates the index by computing the domain and key expressions.

Constraints The expanded QName \$name must be the name of a resource in the statically known indexes. Available indexes must not provide a mapping for the expanded QName \$name.

Semantics Inserts the mapping from QName \$name to an empty map of index entries into available collections. After that the map of index entries is populated as follows:

The domain expression is evaluated first. The result of this step is a sequence of nodes to be indexed, called the *domain sequence*. Let \$domain be this domain sequence. For each domain node in the domain sequence \$domain, the index key expressions are evaluated in some implementation dependent order. In each iterator, the domain node is bound to the variable specified in the index declaration. For the evaluation of each key expression, this variable is bound and is the only free variable that can be accessed. Let D_i be the i -th domain node, and K_{ij} be the key item computed for D_i by the j -th index key specification.

For each $i \in 1 \dots N$ the mapping (also called index entry) $K_{i1} \dots K_{iM} \rightarrow D_i$ where N is the size of the domain sequence and M is the number of key specifications is processed as follows:

- If the index is declared as **unique**, it is a one-to-one map between key tuples and domain items. In this case, if the index already contains an entry whose key is equal to $K_{i1} \dots K_{iM}$, an error is raised err:XQDDY028. Otherwise, the entry $K_{i1} \dots K_{iM} \rightarrow D_i$ is inserted into the map of index entries of the available indexes.
- If the index is **non-unique**, it maps key tuples to sets of domain items. In this case, if the index already contains an entry whose key is equal to $K_{i1} \dots K_{iM}$, then D_i is added to the set associated with $K_{i1} \dots K_{iM}$. Otherwise, the entry $K_{i1} \dots K_{iM} \rightarrow D_i$ is inserted in the index.

- If the index is declared as ordered, then its entries are sorted by the values of their key tuples. Otherwise, its entries are not sorted. Note, typically, an ordered index is implemented by some kind of tree structure (e.g., BTree), whereas an unordered index is implemented by some kind of hash table.
- Comparison of key tuples is done the same way as comparison of flwor tuples by an orderby clause in a flwor expression.

Note that the order of nodes as returned by the `probe-index-range` and `probe-index-point` functions (see 6.2.4) is not guaranteed to be the same as in the domain sequence. Especially, nodes are not returned in document order since document order is not defined among nodes of a collection.

7.1.10 `upd:deleteIndex`

Parameters

```
upd:deleteIndex (
  $name as xs:QName
)
```

Summary Deletes the mapping with QName \$name from the map of available indexes.

Constraints Available indexes must provide a mapping for the expanded QName \$name.

Semantics Deletes the mapping for QName \$name from the map of available indexes.

7.1.11 `upd:refreshIndex`

Parameters

```
upd:refreshIndex (
  $name as xs:QName
)
```

Summary Updates the map of index entries for the entry with QName \$name in the map of available indexes.

Constraints Available indexes must provide a mapping for the expanded QName \$name. The maintenance property for the index with QName \$name is **explicitly maintained**.

Semantics The result of applying the refreshIndex update primitive is the same as applying the following update primitives:

- `upd:deleteIndex($name)`
- `upd:createIndex($name)`

7.2 Update Routines

7.2.1 `upd:mergeUpdates`

The specification of `upd:mergeUpdates` is identical to the specification of `upd:mergeUpdates` in XQuery Update Facility 1.0, except Rule 2 of Semantics is changed as follows:

Optionally, `upd:mergeUpdates` may raise a dynamic error if any of the following conditions are detected:

- Two or more `upd:rename` primitives on the merged list have the same target node **err:XUDY0015**.
- Two or more `upd:replaceNode` primitives on the merged list have the same target node **err:XUDY0016**.
- Two or more `upd:replaceValue` primitives on the merged list have the same target node **err:XUDY0017**.
- Two or more `upd:replaceElementContent` primitives on the merged list have the same target node **err:XUDY0017**.

- Two or more `upd:put` primitives on the merged list have the same `$uri` operand `err:XUDY0031`.
- Two or more primitives on the merged list create conflicting namespace bindings for the same element node `err:XUDY0024`. The following kinds of primitives create namespace bindings:
 - `upd:insertAttributes` creates one namespace binding on the `$target` element corresponding to the implied namespace binding of the name of each attribute node in `$content`.
 - `upd:replaceNode` creates one namespace binding on the `$target` element corresponding to the implied namespace binding of the name of each attribute node in `$replacement`.
 - `upd:rename` creates a namespace binding on `$target`, or on the parent (if any) of `$target` if `$target` is an attribute node, corresponding to the implied namespace binding of `$newName`.
- Two or more `upd:createCollection` primitives on the merged list have the same `$name` operator `err:XQDDY023`
- Two or more `upd:createIndex` primitives on the merged list have the same `$name` operator `err:XQDDY024`
- Two or more `upd:createIntegrityConstraint` primitives on the merged list have the same `$name` operator `err:XQDDY025`

7.2.2 `upd:applyUpdates`

The specification of `upd:applyUpdates` is identical to the specification of `upd:applyUpdates` in XQuery Update Facility 1.0, except Rule 1 and 2 of Semantics are changed as follows:

- Checks the update primitives on `$pul` for compatibility. Raises a dynamic error if any of the following conditions are detected:
 - Two or more `upd:rename` primitives on `$pul` have the same target node `err:XUDY0015`.
 - Two or more `upd:replaceNode` primitives on `$pul` have the same target node `err:XUDY0016`.
 - Two or more `upd:replaceValue` primitives on `$pul` have the same target node `err:XUDY0017`.
 - Two or more `upd:replaceElementContent` primitives on `$pul` have the same target node `err:XUDY0017`.
 - Two or more `upd:put` primitives on `$pul` have the same `$uri` operand `err:XUDY0031`.
 - Two or more primitives on `$pul` create conflicting namespace bindings for the same element node `err:XUDY0024`. The following kinds of primitives create namespace bindings:
 - * `upd:insertAttributes` creates one namespace binding on the `$target` element corresponding to the implied namespace binding of the name of each attribute node in `$content`.
 - * `upd:replaceNode` creates one namespace binding on the `$target` element corresponding to the implied namespace binding of the name of each attribute node in `$replacement`.
 - * `upd:rename` creates a namespace binding on `$target`, or on the parent (if any) of `$target` if `$target` is an attribute node, corresponding to the implied namespace binding of `$newName`.
 - Two or more `upd:createCollection` primitives on `$pul` have the same `$name` operator `err:XQDDY023`.
 - Two or more `upd:createIndex` primitives on `$pul` have the same `$name` operator `err:XQDDY024`.
 - Two or more `upd:createIntegrityConstraint` primitives on `$pul` have the same `$name` operator `err:XQDDY025`.
- The semantics of all update primitives on `$pul`, other than `upd:put` primitives, are made effective in the following order:
 - First, all `upd:insertInto`, `upd:insertAttributes`, `upd:replaceValue`, and `upd:rename` primitives are applied.

- Next, all upd:insertBefore, upd:insertAfter, upd:insertIntoAsFirst, and upd:insertIntoAsLast primitives are applied.
 - Next, all upd:replaceNode primitives are applied.
 - Next, all upd:replaceElementContent primitives are applied.
 - Next, all upd:delete primitives are applied.
 - Next, all upd:createCollection primitives are applied
 - Next, all upd:insertNodesFirst, upd:insertNodesLast, upd:insertNodesBefore, upd:insertNodesAfter primitives are applied.
 - Next, all upd:deleteNode primitives are applied.
 - Next, all upd:deleteCollection primitives are applied.
 - Next, all upd:createIndex primitives are applied.
 - Next, all upd:refreshIndex primitives are applied.
 - Next, all upd:createIntegrityConstraint primitives are applied.
- The semantics of applyUpdates are also modified to include synchronizing indexes and checking integrity constraints.

Note that for each index in the map of available indexes whose maintenance property is **automatically maintained**, the map of index entries is updated in some implementation dependant way. The result is equivalent to applying the upd:refreshIndex update primitive at the same time as this update primitive is applied in the list above.

A Limitations

- moving nodes between collections
- stable property for indexes
- synchronous collections and indexes
- no support utility functions (e.g. import)
- no integrity constraints on types
- no general comparison indexes

A Error Conditions

A.1 Static Errors

A.1.1 err:XQDST001

It is a static error if the expanded QName of the collection is equal (as defined by the eq operator) to the name of another collection in the set of statically known collections.

A.1.2 err:XQDST002

It is a static error if the expanded QName of the index is equal (as defined by the eq operator) to the expanded QName of another index in the set of statically known indexes.

A.1.3 err:XQDST003

It is a static error if the expanded QName of the integrity constraint is equal (as defined by the eq operator) to the name of another integrity constraint in the set of statically known integrity constraints.

A.1.4 err:XQDST004

It is a static error if two or more modifier properties are specified in the declaration of the collection.

A.1.5 err:XQDST005

It is a static error if two or more order properties are specified in the declaration of the collection.

A.1.6 err:XQDST006

It is a static error if inconsistent collection properties are specified in the declaration of the collection.

A.1.7 err:XQDST007

It is a static error if two or more node modifiers are specified in the declaration of the collection.

A.1.8 err:XQDST008

It is a static error if two or more uniqueness properties are specified in the declaration of the index.

A.1.9 err:XQDST009

It is a static error if two or more order properties are specified in the declaration of the index.

A.1.10 err:XQDST010

It is a static error if two or more maintenance properties are specified in the declaration of the index.

A.1.11 err:XQDST011

It is a static error if any expression used in an integrity constraint declaration has dependencies to the dynamic context.

A.1.12 err:XQDST012

It is a static error if an integrity constraint is dependent on non-existing collections (forward references are not allowed).

A.1.13 err:XQDST013

It is a static error if the source collection in an integrity constraint declaration is not declared in the same module as the integrity constraint.

A.1.14 err:XQDST014

It is a static error if the expanded QName of a collection declared in an imported module is equal (as defined by the eq operator) to the expanded QName of a collection declared in the importing module or in another imported module (even if the declarations are consistent).

A.1.15 err:XQDST015

It is a static error if the expanded QName of an index declared in an imported module is equal (as defined by the eq operator) to the expanded QName of an index declared in the importing module or in another imported module (even if the declarations are consistent).

A.1.16 err:XQDST016

It is a static error if the expanded QName of an integrity constraint declared in an imported module is equal (as defined by the eq operator) to the expanded QName of an integrity constraint declared in the importing module or in another imported module (even if the declarations are consistent).

A.1.17 err:XQDST017

Key expression must specify an atomic type other than *xs:untypedAtomic* and specify no or ? as occurrence indicator.

A.1.18 err:XQDST018

The domain expression in an index declaration must not access the context focus.

A.1.19 err:XQDST019

The domain or key expressions in an index declaration must be deterministic.

A.1.20 err:XQDST020

The argument to the `xqddf:collection` call must be a constant expression.

A.2 Dynamic Errors

A.2.1 err:XQDDY001

The modifier property of a collection doesn't allow the function to be called.

A.2.2 err:XQDDY002

The order property of a collection doesn't allow the function to be called.

A.2.3 err:XQDDY003

The expanded QName of the collection is not equal (as defined by the eq operator) to the name of a collection in the statically known collections.

A.2.4 err:XQDDY004

The expanded QName of the index is not equal (as defined by the eq operator) to the name of a collection in the statically known indexes.

A.2.5 err:XQDDY005

The expanded QName of the integrity constraint is not equal (as defined by the eq operator) to the name of a collection in the statically known integrity constraints.

A.2.6 err:XQDDY006

The expanded QName of the collection is not equal (as defined by the eq operator) to the name of a collection in the dynamically available collections.

A.2.7 err:XQDDY007

The expanded QName of the index is not equal (as defined by the eq operator) to the name of an index in the dynamically available indexes.

A.2.8 err:XQDDY008

The expanded QName of the integrity constraint is not equal (as defined by the eq operator) to the name of an integrity constraint in the active integrity constraints.

A.2.9 err:XQDDY009

The set of available collections already provides a mapping for the expanded QName.

A.2.10 err:XQDDY010

The set of available indexes already provides a mapping for the expanded QName.

A.2.11 err:XQDDY011

The set of active integrity constraints already provides a mapping for the expanded QName.

A.3 err:XQDDY012

The sequences in the available collections mapped by the expanded QName doesn't contain any nodes.

A.3.1 err:XQDDY013

An updating expression not allowed on node that belongs to a read-only collection.

A.3.2 err:XQDDY014

Any of the in-scope variables references a node that belongs to the collection.

A.3.3 err:XQDDY015

The domain or key expression of any of the available indexes access the collection.

A.3.4 err:XQDDY016

The target node is not contained in the available collection.

A.3.5 err:XQDDY017

Any of the nodes are not contained in the available collection.

A.3.6 err:XQDDY018

The expression in the integrity constraint must return a valid *xs:boolean* value.

A.3.7 err:XQDDY019

The unique index already contains an entry with equal keys.

A.3.8 err:XQDDY020

The domain sequence of an index must not contain any duplicate nodes.

A.3.9 err:XQDDY021

Each node in a domain expression must belong to a collection.

A.3.10 err:XQDDY022

The domain expression must generate a sequence of nodes.

A.3.11 err:XQDDY023

Two or more `upd:createCollection` primitives on the merged list have the same name.

A.3.12 err:XQDDY024

Two or more `upd:createIndex` primitives on the merged list have the same name.

A.3.13 err:XQDDY025

Two or more `upd:createIntegrityConstraint` primitives on the merged list have the same name.

A.3.14 err:XQDDY026

If `xqddf:index-of` is called on an unordered collection.

A.3.15 err:XQDDY027

Each domain node in the domain sequence of an index must belong to a collection that appear in the available collections of the module that contains the index declaration.

A.3.16 err:XQDDY028

If the map of index entries already contains a mapping for the given key and the index is declared as unique, an error is raised.

A.3.17 err:XQDDY029

The node to delete is not part of the collection.

A.4 Type Errors

A.4.1 err:XQDTY001

The content does not match the expected type according to the rules for `SequenceType Matching`.

A.4.2 err:XQDTY002

During expression evaluation, the result of the key expression must match the given type using the rules of `SequenceType matching`.

References

[XQUERY11] XQuery 1.1 W3C Working Draft 3 December 2008 <http://www.w3.org/TR/xquery-11/>

[XQU10] XQuery Update Facility 1.0 W3C Candidate Recommendation 09 June 2009
<http://www.w3.org/TR/2009/CR-xquery-update-10-20090609/>

[XQFO] XQuery 1.0 and XPath 2.0 Functions and Operators W3C Recommendation 23 January 2007
<http://www.w3.org/TR/xpath-functions/>