JSONiq Extension to XQuery 1.0

Usecases for the JSONiq Extension to XQuery

Jonathan Robie

Ghislain Fourny

Matthias Brantner

Daniela Florescu

Till Westmann

Markos Zaharioudakis

JSONiq Extension to XQuery 1.0 Usecases for the JSONiq Extension to XQuery

Author	Jonathan Robie	jonathan.robie@gmail.com
Author	Ghislain Fourny	ghislain.fourny@inf.ethz.ch
Author	Matthias Brantner	matthias.brantner@oracle.com
Author	Daniela Florescu	dana.florescu@oracle.com
Author	Till Westmann	Till@couchbase.com
Author	Markos Zaharioudakis	markos.zaharioudakis@oracle.com
Editor	Ghislain Fourny	ghislain.fourny@inf.ethz.ch

This document introduces use cases for the JSONiq extension to XQuery.

JSONiq: Use Cases

XQuery for JSON, JSON for XQuery

On the Internet, most data is exchanged and processed using JSON or XML, and displayed using HTML. These languages have a great deal in common: they are based on named structures that can be nested, using a Unicode-based representation that is both human-readable and machine-readable. HTML is optimized for representing web pages, XML is optimized for representing document data, and JSON is optimized for representing programming language structures. In many applications, programmers work with only one of these formats. In many others, programmers work with all three.

JSON has been called a “fat free alternative to XML”[1]. Markup for programming structures does not need XML namespaces, whitespace-handling rules, the distinction between elements and attributes, or a structure designed to support document order. Markup for programming structures does need datatypes. XML can support datatypes via XML Schema, a very complex specification that does not correspond closely to the type system of most programming languages, or it can be used without datatype support. Neither is optimal for programming structures. For traditional programming structures, programming with JSON is significantly easier than programming with XML.

There is currently no standard query language for JSON. XQuery is the standard query language for XML, and has been implemented in databases, streaming processors, data integration platforms, application integration platforms, XML message routing software, web browser plugins, and other environments. JSONiq is a query language based on XQuery; it is designed to allow an existing XQuery processor to be rewritten to support JSON with moderate effort. JSONiq defines a profile called XQ-- that is based on XQuery, removing expressions that support constructing and navigating XML, and adding expressions that support constructing and navigating JSON. Syntax diagrams for XQ-- are available at *http://jsoniq.com/grammars/xq--/ui.xhtml*. Because JSON is so much simpler than XML, XQ-- is much simpler than XQuery, for both users and implementers. XQ-- also makes it much easier to write middleware using JSON as a logical model, because JSON more closely resembles the data typically processed in middleware, and allows a much simpler type system.

But programmers who work with JSON often work with XML and HTML as well. In some applications, XML is used because it can better represent document-oriented data such as medical records, legal documents, pharmaceutical trial data, or news feeds. In other applications, standards or existing software systems may require data to be produced or consumed as XML. Many applications require data to be converted to HTML formats; some use HTML documents as source data. To support such applications, JSONiq defines a profile called XQ++ that includes the complete XQuery language, adding the same JSON extensions found in XQ--. Syntax diagrams for XQ++ are available at *http://jsoniq.com/grammars/xq++/ui.xhtml*.

JSONiq is useful for applications like these:

- Queries and Transformations for JSON

  JSONiq can be used for queries on JSON that return JSON results. These queries have the full power of XQuery, including FLWOR expressions, grouping, windowing, updates, and full-text. But queries or implementations that do not use XML or produce XML can use a subset of XQuery, and need not deal with the additional complexity that XML requires. Used in this way, JSONiq adds significant value to NoSQL databases.

  Examples of these queries can be found in *Section 1.1.1, “Joins”*, *Section 1.1.2, “Grouping Queries for JSON”*, and *Section 1.1.3, “JSON to JSON Transformations”*.

[1] See *JSON: The Fat-Free Alternative to XML* [http://www.json.org/fatfree.html].

- JSON Updates

  JSONiq provides updates for JSON, in the same way that the XQuery Update Facility provides updates for XML.

  An example of such a query can be found in *Section 1.1.9, "JSON Updates"*.

- JSON to XML transformations

  The most stubborn JSON programmer is sometimes forced to use XML-based formats. JSONiq makes it easy to create XML by querying JSON data.

  An example of such a query can be found in *Section 1.1.5, "Transforming JSON to SVG"*.

- XML to JSON transformations

  XQuery hides much of the complexity of XML, and simplifies the process of converting XML to JSON, while also providing a full featured query language to create any JSON result desired.

  An example of such a query can be found in *Section 1.1.4, "Converting XML to JSON"*.

- JSON to HTML transformations

  JSON is often used for data that is displayed in HTML. JSONiq makes it easy to transform JSON data into various views, creating the HTML needed to display results.

  An example of such a query can be found in *Section 1.1.6, "Transforming Arrays to HTML Tables"*.

- Windowing and input sequences

  XQuery provides sliding windows and tumbling windows for processing events and input sequences.

  An example of such a query can be found in *Section 1.1.7, "Windowing Queries"*.

- JSON views in middleware

  XQuery is used in middleware systems to provide XML views of data sources. Because JSON is simpler than XQuery, JSON-based views are an attractive alternative to XML-based views in applications that use large scale relational, object, or semi-structured data. JSONiq provides a powerful query language for systems that provide such views.

  An example of such a query can be found in *Section 1.1.8, "JSON views in middleware"*.

- Adding missing functionality to XQuery

  JSON Objects and JSON Arrays are extremely convenient for intermediate results in XQuery programs, even if they query XML to create XML.

  XQuery currently has no way to create complex data structures without copying and atomization, and no way to nest Arrays. Objects and Arrays may be particularly useful for applications like:

  - Function parameters
  - Returning results from functions
  - Indexes
  - Maps of function items

- Representing internal XQuery structures such as tuple streams or Pending Update Lists.

1.1. Sample Queries

1.1.1. Joins

The following queries are based on a social media site that allows users to interact with their friends. **`collection("users")`** contains data on users and their friends:

```
{
  "name" : "Sarah",
  "age" : 13,
  "gender" : "female",
  "friends" : [ "Jim", "Mary", "Jennifer"]
}

{
  "name" : "Jim",
  "age" : 13,
  "gender" : "male",
  "friends" : [ "Sarah" ]
}
```

The following query performs a join on Sarah's friend list to return the Object representing each of her friends:

```
for $sarah in collection("users")
    $friend in collection("users")
where $sarah("name") = "Sarah"
  and (some $name in libjn:members($sarah("friends"))
       satisfies $friend("name") = $name)
return $friend
```

The query can be simplified using a filter. In the following expression, **`[.("name") = "Sarah"]`** is a filter that restricts the set of users to the one named "Sarah". The following query returns the same results as the previous one:

```
let $sarah := collection("users")[.("name") = "Sarah"]
for $friend in libjn:members($sarah("friends"))
return collection("users")[.("name") = $friend]
```

1.1.2. Grouping Queries for JSON[2]

collection("sales") is an unordered sequence that contains the following objects:

```
{ "product" : "broiler", "store number" : 1, "quantity" : 20  },{ "product" : "toaster",
 "store number" : 2, "quantity" : 100 },{ "product" : "toaster", "store number" :
 2, "quantity" : 50 },{ "product" : "toaster", "store number" : 3, "quantity" :
 50 },{ "product" : "blender", "store number" : 3, "quantity" : 100 },{ "product" :
 "blender", "store number" : 3, "quantity" : 150 },{ "product" : "socks", "store
 number" : 1, "quantity" : 500 },{ "product" : "socks", "store number" : 2, "quantity" :
 10 },{ "product" : "shirt", "store number" : 3, "quantity" : 10 }
 20  },{ "product" : "toaster", "store number" : 2, "quantity"
```

[2] These queries are based on similar queries in the XQuery 3.0 Use Cases.

```
: 100 },{ "product" : "toaster", "store number" : 2, "quantity"
: 50 },{ "product" : "toaster", "store number" : 3, "quantity"
: 50 },{ "product" : "blender", "store number" : 3, "quantity"
: 100 },{ "product" : "blender", "store number" : 3, "quantity"
: 150 },{ "product" : "socks", "store number" : 1, "quantity"
: 500 },{ "product" : "socks", "store number" : 2, "quantity"
: 10 },{ "product" : "shirt", "store number" : 3, "quantity"
```

We want to group sales by product, across stores.

Query:

```
jn:object(
  for $sales in collection("sales")
  let $pname := $sales("product")
  group by $pname
  return $pname : sum($sales("quantity"))
)
```

Result:

```
{
  "blender" : 250,
  "broiler" : 20,
  "shirt" : 10,
  "socks" : 510,
  "toaster" : 200
}
```

Now let's do a more complex grouping query, showing sales by category within each state. We need further data to describe the categories of products and the location of stores.

collection("products") contains the following data:

```
{ "name" : "broiler", "category" : "kitchen", "price" : 100, "cost" : 70 },{ "name" :
 "toaster", "category" : "kitchen", "price" : 30, "cost" : 10 },{ "name" : "blender",
 "category" : "kitchen", "price" : 50, "cost" : 25 },{ "name" : "socks", "category" :
 "clothes", "price" : 5, "cost" : 2 },{ "name" : "shirt", "category" : "clothes", "price" :
 10, "cost" : 3 }
 70 },{ "name" : "toaster", "category" : "kitchen", "price" : 30, "cost" :
 10 },{ "name" : "blender", "category" : "kitchen", "price" : 50, "cost" :
 25 },{ "name" : "socks", "category" : "clothes", "price" : 5, "cost" :
 2 },{ "name" : "shirt", "category" : "clothes", "price" : 10, "cost" :
```

collection("stores") contains the following data:

```
{ "store number" : 1, "state" : CA },{ "store number" : 2, "state" : CA },{ "store number" :
 3, "state" : MA },{ "store number" : 4, "state" : MA }
},{ "store number" : 2, "state" : CA
},{ "store number" : 3, "state" : MA
},{ "store number" : 4, "state" : MA
```

The following query groups by state, then by category, then lists individual products and the sales associated with each.

Query:

```
jn:object(
  for $store in collection("stores")
  let $state := $store("state")
```

```
  group by $state
  return {
    $state : jn:object(
      for $product in collection("products")
      let $category := $product("category")
      group by $category
      return {
        $category : jn:object(
          for $sales in collection("sales")
          where $sales("store number") = $store("store number")
            and $sales("product") = $product("name")
          let $pname := $sales("product")
          group by $pname
          return $pname : sum( $sales("quantity") )
        )
      }
    )
  }
)
```

Result:

```
{
  "CA" : {
      "clothes" : {
         "socks" :  510
      },
      "kitchen" : {
         "broiler" : 20,
         "toaster" : 150
      }
  },
  "MA" : {
      "clothes" : {
         "shirt" : 10
       },
      "kitchen" : {
         "blender" : 250,
         "toaster" : 50
      }
  }
}
```

1.1.3. JSON to JSON Transformations

The following query takes satellite data, and summarizes which satellites are visible. The data for the query is a simplified version of a Stellarium file that contains this information.

Data:

```
{
  "creator" : "Satellites plugin version 0.6.4",
  "satellites" : {
    "AAU CUBESAT" : {
      "tle1" : "1 27846U 03031G 10322.04074654  .00000056  00000-0  45693-4 0  8768",
      "visible" : false
    },
    "AJISAI (EGS)" : {
      "tle1" : "1 16908U 86061A 10321.84797408 -.00000083  00000-0  10000-3 0  3696",
      "visible" : true
    },
    "AKARI (ASTRO-F)" : {
      "tle1" : "1 28939U 06005A 10321.96319841  .00000176  00000-0  48808-4 0  4294",
```

```
      "visible" : true
    }
  }
}
```

We want to query this data to return a summary that looks like this.

Result:

```
{
  "visible" : [
     "AJISAI (EGS)",
     "AKARI (ASTRO-F)"
  ],
  "invisible" : [
     "AAU CUBESAT"
  ]
}
```

The following is a JSONiq query that returns the desired result.

Query:

```
let $sats := jn:json-doc("satellites.json")("satellites")return {  "visible" : [     for $sat
 in jn:keys($sats)     where $sats($sat)("visible")     return $sat  ],  "invisible" : [
 for $sat in jn:keys($sats)     where not($sats($sat)("visible"))     return $sat  ]}
lites")return
{  "visible" :
[     for $sat in
jn:keys($sats)     where $sats($sat)
("visible")     return
$sat
],  "invisible" :
[     for $sat in
jn:keys($sats)     where not($sats($sat)
("visible"))     return
$sat
]
```

1.1.4. Converting XML to JSON

JSON programmers frequently need to convert XML to JSON. The following query is based on a Wikipedia XML export format, using data from the category "Origami". Here is an excerpt of this data:

Data:

```
<mediawiki>  <siteinfo>    <sitename>Wikipedia</sitename>    <page>      <title>Kawasaki's
 theorem</title>      <id>14511776</id>      <revision>        <id>435519187</id>
 <timestamp>2011-06-21T20:08:56Z</timestamp>        <contributor>          <username>Some
 jerk on the Internet</username>          <id>6636894</id>        </contributor>!!! SNIP !!!
    <page>      <title>Origami techniques</title>      <id>193590</id>      <revision>
  <id>447687387</id>        <timestamp>2011-08-31T17:21:49Z</timestamp>        <contributor>
          <username>Dmcq</username>          <id>3784322</id>        </contributor>!!!
 SNIP !!!    <page>      <title>Mathematics of paper folding</title>      <id>232840</id>
   <revision>        <id>440970828</id>        <timestamp>2011-07-23T09:10:42Z</timestamp>
      <contributor>          <username>Tabletop</username>          <id>173687</id>        </
contributor>
diawiki>
  <siteinfo>

 <sitename>Wikipedia</sitename>
  <page>      <title>Kawasaki's
```

```
 theorem</title>
 <id>14511776</id>
  <revision>
 <id>435519187</id>
 <timestamp>2011-06-21T20:08:56Z</timestamp>
  <contributor>          <username>Some jerk on the
 Internet</username>
 <id>6636894</id>

 </contributor>!!!

SNIP !!!
  <page>      <title>Origami
 techniques</title>
 <id>193590</id>
  <revision>
 <id>447687387</id>
 <timestamp>2011-08-31T17:21:49Z</timestamp>
  <contributor>
 <username>Dmcq</username>
 <id>3784322</id>

 </contributor>!!!

SNIP !!!
  <page>      <title>Mathematics of paper
 folding</title>
 <id>232840</id>
  <revision>
 <id>440970828</id>
 <timestamp>2011-07-23T09:10:42Z</timestamp>
  <contributor>
 <username>Tabletop</username>
 <id>173687</id>
```

The following query converts this data to JSON:

Query:

```
for $page in doc("Wikipedia-Origami.xml")//page
return {
  "title": string($page/title),
  "id" : string($page/id),
  "last updated" : string($page/revision[1]/timestamp),
  "authors" : [
       for $a in $page/revision/contributor/username
       return string($a)
  ]
}
```

Result:

```
{
  "title" : "Kawasaki's theorem",
  "id" : "14511776",
  "last updated" : "2011-06-21T20:08:56Z",
  "timestamp" : "2011-06-21T20:08:56Z",
  "authors" : ["Some jerk on the Internet" ]
},
{
  "title" : "Origami techniques",
  "id" : "193590",
  "last updated" : "2011-08-31T17:21:49Z",
  "timestamp" : "2011-08-31T17:21:49Z",
  "authors" : ["Dmcq" ]
```

```
},
{
  "title" : "Mathematics of paper folding",
  "id" : "232840",
  "last updated" : "2011-07-23T09:10:42Z",
  "timestamp" : "2011-07-23T09:10:42Z",
  "authors" : ["Tabletop" ]
}
```

1.1.5. Transforming JSON to SVG

Suppose a JavaScript implementation provides an interface for JSONiq queries, and a JavaScript program contains the following data [3]:

```
var data = {   "color" : "blue",   "closed" : true,   "points" : [[10,10], [20,10], [20,20],
 [10,20]]   };
{   "color" :
"blue",   "closed" :
true,   "points" : [[10,10], [20,10], [20,20],
[10,20]]
```

This data can be converted to SVG by placing the text of a query in a JavaScript variable and calling the appropriate JavaScript function to invoke the query:

```
var query =
 "declare variable $input external;
  declare variable $stroke := attribute $stroke { $input("color") };
  declare variable $points := attribute $points { jn:flatten($input("points")) };
  if ($input("closed")) then
    <svg><polygon>{ $stroke, $points }</polygon></svg>
  else
    <svg><polyline>{ $stroke, $points }</polyline></svg>"
```

This query can be invoked with a JavaScript API call:

```
jsoniq(data, query)
```

Here is the result of the above query:

```
<svg><polygon stroke="blue" points="10 10 20 10 20 20 10 20" /></svg>
```

1.1.6. Transforming Arrays to HTML Tables

The data in a JSON array is frequently displayed using HTML tables. The following query shows how to transform from the former to the latter.

The following Object contains the labels desired for columns and rows, as well as the data for the table.

```
{
  "col labels" : ["singular", "plural"],
  "row labels" : ["1p", "2p", "3p"],
```

[3] This example is based on an example on Stefan Goessner's JSONT site (*http://goessner.net/articles/jsont/*).

```
  "data" :
     [
        ["spinne", "spinnen"],
        ["spinnst", "spinnt"],
        ["spinnt", "spinnen"]
     ]
}
```

The following query creates an HTML table, using the column headings and row labels as well as the data in the Object shown above.

```
<table>  <tr> (: Column headings :)  {     <th> </th>,     for $th in jn:members((jn:json-
doc("table.json")("col labels")))     return <th>{ $th }</th>  }  </tr>  {  (: Data for each
 row :)     for $r at $i in jn:members((jn:json-doc("table.json")("data")))     return
  <tr>         {           <td>{ jn:members(jn:json-doc("table.json")("row labels")($i)) }</
td>,           for $c in jn:members($r)           return <td>{ $c }</td>         }        </
tr>  }</table>
ble>  <tr> (: Column headings
:)
{     <th> </
th>,     for $th in jn:members((jn:json-doc("table.json")("col
labels")))     return <th>{ $th }</
th>
}  </
tr>  {  (: Data for each row
:)     for $r at $i in jn:members((jn:json-doc("table.json")
("data")))
return
<tr>
{           <td>{ jn:members(jn:json-doc("table.json")("row labels")($i)) }</
td>,           for $c in
jn:members($r)           return <td>{ $c }</
td>
}        </
tr>
}</
```

1.1.7. Windowing Queries

XQuery provides support for both sliding windows and tumbling windows, frequently used to analyze event streams or other sequential data. This simple windowing example converts a sequence of items to a table with three columns (using as many rows as necessary), and assigns a row number to each row.

Data:

```
[
  { "color" : "Green" },
  { "color" : "Pink" },
  { "color" : "Lilac" },
  { "color" : "Turquoise" },
  { "color" : "Peach" },
  { "color" : "Opal" },
  { "color" : "Champagne" }
]
```

Query:

```

<table>{
  for tumbling window $w in jn:members(jn:json-doc("colors.json"))
    start at $x when fn:true()
```

```
    end at $y when $y - $x = 2
  return
    <tr>{
      for $i in $w
      return
        <td>{ $i }</td>
    }</tr>
}</table>
```

Result:

```
<table>  <tr>    <td>Green</td>    <td>Pink</td>    <td>Lilac</td>  </tr>  <tr>
 <td>Turquoise</td>    <td>Peach</td>    <td>Opal</td>  </tr>  <tr>    <td>Champagne</td>  </
tr></table>
ble>
<tr>    <td>Green</
td>    <td>Pink</
td>    <td>Lilac</
td>  </
tr>
<tr>    <td>Turquoise</
td>    <td>Peach</
td>    <td>Opal</
td>  </
tr>
<tr>    <td>Champagne</
td>  </
tr></
```

1.1.8. JSON views in middleware

This example assumes a middleware system that presents relational tables as JSON arrays. The following two tables are used as sample data.

Table 1.1. Users

userid	firstname	lastname
W0342	Walter	Denisovich
M0535	Mick	Goulish

The JSON representation this particular implementation provides for the above table looks like this:

```
[
  { "userid" : "W0342", "firstname" : "Walter", "lastname" : "Denisovich" },
  { "userid" : "M0535", "firstname" : "Mick", "lastname" : "Goulish" }
]
```

Table 1.2. Holdings

userid	ticker	shares
W0342	DIS	153212312
M0535	DIS	10
M0535	AIG	23412

The JSON representation this particular implementation provides for the above table looks like this:

```
[
  { "userid" : "W0342", "ticker" : "DIS", "shares" : 153212312 },
  { "userid" : "M0535", "ticker" : "DIS", "shares" : 10 },
```

```
  { "userid" : "M0535", "ticker" : "AIG", "shares" : 23412 }
]
```

The following query uses the fictitious vendor's `vendor:table()` function to retrieve the values from a table, and creates an Object for each user, with a list of the user's holdings in the value of that Object.

```
[
  for $u in vendor:table("Users")
  order by $u("userid")
  return jn:object(
    libjn:project($u, "userid"),
    {
      "first" :  $u("firstname"),
      "last" :   $u("lastname"),
      "holdings" : [
        for $h in vendor:table("Holdings")
        where $h("userid") = $u("userid")
        order by $h("ticker")
        return jn:object(
          libjn:project($h, "ticker"),
          { "share" : $h("shares") }
        )
      ]
    }
  )
]
```

1.1.9. JSON Updates

The XQuery Update Facility allows XML data to be updated. JSONiq provides updating functions to allow JSON to be updated.

Suppose an application receives an order that contains a credit card number, and needs to put the user on probation.

Data for an order:

```
{
  "user" : "Deadbeat Jim",
  "credit card" : VISA 4111 1111 1111 1111,
  "product" : "lottery tickets",
  "quantity" : 243
}
```

collection("users") contains the data for each individual user:

```
{
  "name" : "Deadbeat Jim",
  "address" : "1 E 161st St, Bronx, NY 10451",
  "risk tolerance" : "high"
}
```

The following query adds **`"status" : "credit card declined"`** to the user's record.

```
let $dbj := collection("users")[ .("name") = "Deadbeat Jim" ]return insert json { "status" :
 "credit card declined" } into $dbj
]return insert json { "status" : "credit card declined" } into
```

After the update is finished, the user's record looks like this:

```
{
  "name" : "Deadbeat Jim",
  "address" : "1 E 161st St, Bronx, NY 10451",
  "status" : "credit card declined",
  "risk tolerance" : "high"
}
```

1.1.10. Data Transformations

Many applications need to modify data before forwarding it to another source. The XQuery Update Facility provides an expression called a tranform expression that can be used to create modified copies. The transform expression uses updating expressions to perform a transformation. JSONiq defines updating functions for JSON, which can be used in the XQuery transform expression.

Suppose an application make videos available using feeds from Youtube. The following data comes from one such feed:

```
{
    "encoding" : "UTF-8",
    "feed" : {
        "author" : [
            {
                "name" : {
                    "$t" : "YouTube"
                },
                "uri" : {
                    "$t" : "http://www.youtube.com/"
                }
            }
        ],
        "category" : [
            {
                "scheme" : "http://schemas.google.com/g/2005#kind",
                "term" : "http://gdata.youtube.com/schemas/2007#video"
            }
        ],
        "entry" : [
            {
                "app$control" : {
                    "yt$state" : {
                        "$t" : "Syndication of this video was restricted by its owner.",
                        "name" : "restricted",
                        "reasonCode" : "limitedSyndication"
                    }
                },
                "author" : [
                    {
                        "name" : {
                            "$t" : "beyonceVEVO"
                        },
                        "uri" : {
                            "$t" : "http://gdata.youtube.com/feeds/api/users/beyoncevevo"
                        }
                    }
                ]
!!! SNIP !!!
```

The following query creates a modified copy of the feed by removing all entries that restrict syndication.

```
let $feed := jn:json-doc("incoming.json")
return
```

```
copy $out := $feed
modify
   let $feed := $out("feed")
   let $feed-entry := $feed("entry")
   for $entry at $pos in jn:members( $feed-entry )
   where $entry("app$control")("yt$state")("name") = "restricted"
   return delete json $feed-entry($pos)
return $out
```

Appendix A. Revision History

Revision 0-0 Thu Apr 4 2013 **Dude McPants** *Dude.McPants@example.com*

Initial creation of book by publican

Index