JSONiq: XQuery for JSON

Jonathan Robie
Matthias Brantner
Daniela Florescu
Ghislain Fourny
Till Westmann

Abstract

XML and JSON have become the dominant formats for exchanging data on the Internet, and applications frequently need to send and receive data in many different JSON-based or XML-based formats. For XML data, a query language like XQuery can be used to query data, create or update data, transform it from one format to another, or route data. Adding JSON support to XQuery allows it to perform these tasks for both XML and JSON, combining data from multiple sources as needed.

JSONiq is a query language for JSON, based on XQuery. It is designed to allow an existing XQuery processor to be rewritten to support JSON with moderate effort. One profile of JSONiq removes everything directly related to XML, adding JSON constructors and navigation. Another profile of JSONiq includes the full XQuery language, with added JSON support, allowing queries to consume or produce JSON, XML, or HTML.

1. Introduction

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 makes it easy for an XQuery processor to support JSON in these same environments.

Currently, there is no standard query language for JSON, and many JSON programmers do not want the complexity of XML. Because of the modular, compositional design of XQuery, it is easy to remove the XML-specific portions of the language, and to add similar functionality for JSON. XQuery consists largely of XML constructors, XML path expressions, and expressions for querying and transforming data, such as FLWOR expressions, that are not directly based on XML. One profile of JSONiq removes everything directly related to XML, adding JSON constructors and navigation. The resulting language is simpler and easier to optimize, and well suited to JSON views in middleware. But it includes sophisticated query capabilities, including grouping and windowing, that are very useful in a JSON environment.

Another profile of JSONiq includes the full XQuery language, with added JSON support, allowing queries to consume or produce JSON, XML, or HTML. XML continues to be widely used for data interchange on the Internet, and many applications need to process both JSON and XML. XML has significant advantages for document data, is well supported by standards, is part of a rich ecosystem of tools, libraries, and language extensions, and is required by many existing data interchange standards. Particularly in applications where data resembles human documents, including many healthcare, financial, government, intelligence, legal, and publishing applications, XML continues to have advantages over JSON — particularly when document data needs to be queried. Adding the JSONiq extensions to XQuery allows queries to process or produce JSON, XML, or HTML, combining and transforming data from any of these formats as needed. And adding JSON objects and arrays to XQuery also allows provides these useful data structures to programs that process only XML.

2. JSONiq in a Nutshell

JSONiq adds constructors for creating JSON objects and arrays, and member accessors for navigating them. JSON constructors look like JSON objects and arrays. For instance, the following query creates a JSON object for a social media site:

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

```
}
```

JSONiq member accessors navigate JSON objects using the names of name/value pairs or the position of array items. For instance, if the above object is bound to the variable **`$sarah`**, then **`$sarah("age")`** returns **`13`**.

JSONiq allows expressions in JSON constructors in the same way that XQuery allows expressions in XML constructors. The following JSONiq query creates a new user named "Jennifer", one year older than Sarah, with a friend list based on Sarah's. Jennifer does not appear on her own friends list, but Sarah does:

```
let $sarah := collection("users")[.("name") = "Sarah"]
return {
    "name" : "Jennifer",
    "age"  : $sarah("age") + 1,
    "friends" : [ values($sarah("friends")) except "Jennifer", "Sarah" ]
}
```

The result of the above query is:

```
{
    "name" : "Jennifer",
    "age"  : 14,
    "friends" : [ "Jim", "Mary", "Sarah"]
}
```

JSONiq also adds a few functions, including updating functions. But most of the power of JSONiq comes from the existing XQuery language, which is well designed for transformations on hierarchical structures, well understood, and widely implemented.

3. Grouping Queries for JSON

JSONiq allows the same functionality for JSON that XQuery provides for XML, except for functionality that depends directly on the properties of XML. This includes joins, grouping, and windowing. This section demonstrates this using a grouping example based on a similar example in the XQuery 3.0 Use Cases.

Suppose 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 }
```

We want to group sales by product, across stores.

Query:

```
{
  for $sales in collection("sales")
  let $pname := $sales("product")
  group by $pname
  return $pname : sum(for $s in $sales return $s("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 }
```

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 }
```

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

Query:

```
{
  for $store in collection("stores")
  let $state := $store("state")
  group by $state
  return
     $state : {
       for $product in collection("products")
       let $category := $product("category")
       group by $category
       return
         $category : {
            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( for $s in $sales return $s("quantity") )
         }
     }
}
```

Result:

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

4. 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.

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

Table 1. Users

userid	firstname	lastname
W0342	Walter	Denisovich

userid	firstname	lastname
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 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 {
    "userid" : $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 {
            "ticker" : $u("ticker"),
            "share" : $u("shares")
         }
    ]
  }
]
```

5. JSON with XML and HTML

When JSONiq is used together with the full XQuery language, it can be used to convert data from one format to another, whether the formats use JSON, XML, or HTML. It can also be used to combine data from multiple formats, transform it, and create a result in any desired format.

For instance, suppose the following JSON data needs to be converted to HTML:

```
{
  "col labels" : ["singular", "plural"],
  "row labels" : ["1p", "2p", "3p"],
  "data" :
     [
        ["spinne", "spinnen"],
        ["spinnst", "spinnt"],
        ["spinnt", "spinnen"]
     ]
}
```

The following query creates an HTML table from this JSON Object, using the column headings and row labels specified:

```
<table>
  <tr> (: Column headings :)
     {
```

```
        <th> </th>,
        for $th in json("table.json")("col labels")
        return <th>{ $th }</th>
     }
  </tr>
  {  (: Data for each row :)
     for $r at $i in json("table.json")("data")
     return
        <tr>
         {
           <th>{ json("table.json")("row labels")[$i] }</th>,
           for $c in $r
           return <td>{ $c }</td>
         }
        </tr>
  }
</table>
```

The result of the query is the following HTML table:

```
<table>
   <tr>
     <th> </th>
     <th>singular</th>
     <th>plural</th>
   </tr>
   <tr>
     <th>1p</th>
     <td>spinne</td>
     <td>spinnen</td>
   </tr>
   <tr>
     <th>2p</th>
     <td>spinnst</td>
     <td>spinnt</td>
   </tr>
   <tr>
     <th>3p</th>
     <td>spinnt</td>
     <td>spinnen</td>
   </tr>
</table>
```

6. Conclusion

JSON and Cloud-based computing have brought new challenges to the Internet. XML is no longer the universal data interchange format once envisioned, but it is still widespread and XML-based services need to be used together with JSON-based services and various other data sources.

XQuery is a powerful and mature query language that has been implemented in many environments. By adding a small number of constructors and member accessors for JSON objects and arrays, JSONiq makes it possible for queries to consume, combine, or produce data in any JSON, XML, or HTML format, converting among them as needed. Any existing XQuery processor can add JSON support with moderate effort. Data integration was always an important application for XQuery, and adding support for JSON allows XQuery to play well with both of the dominant data interchange formats on the Internet. Where XML-based views of data such as relational databases are currently used, JSON-based views provide a simpler alternative.

In some implementations and environments, both JSON and XML will be supported. In other environments, only JSON support will be provided for the sake of simpler, more easily optimizable implementations.

Bibliography

[JSONiq] *JSONiq Language Specification*[1]. Jonathan Robie. Matthias Brantner. Daniela Florescu. Ghislain Fourny.

[JSONiq Use Cases] *JSONiq: XQuery for JSON, JSON for XQuery*[2]. Jonathan Robie. Matthias Brantner. Daniela Florescu. Ghislain Fourny.

[XQuery 3.0] *XQuery 3.0: An XML Query Language. W3C Working Draft, 14 June 2011*[3]. World Wide Web Consortium . 16 November 1999.

[1] http://jsoniq.com/docs/spec/en-US/html/index.html
[2] http://jsoniq.com/docs/spec/en-US/html/index.html
[3] www.w3.org/TR/xquery-30/


DRAFT