JSONiq - the SQL of NoSQL 1.0

JSONiq Use Cases

Use cases for the JSONiq core language.

Jonathan Robie

Ghislain Fourny

Matthias Brantner

Daniela Florescu

Till Westmann

Markos Zaharioudakis

JSONiq - the SQL of NoSQL 1.0 JSONiq Use Cases
Use cases for the JSONiq core language.
Edition 1.0.9

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 some use cases for the JSONiq core language.

These use cases were originally use cases for the JSONiq extension to XQuery (this is where it all started). They were converted to the more pleasant JSONiq core syntax.

The queries where results are supplied were executed with Zorba.

JSONiq Use Cases

- 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"*.

- 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 1.1.4, "JSON Updates"*.

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:

> Example 1.1. A join query.
>
> ```
> for $sarah in collection("users"),
>     $friend in collection("users")
> where $sarah.name eq "Sarah"
>       and
>       (some $name in $sarah.friends[]
>        satisfies $friend.name eq $name)
> return $friend
> ```

Result:

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

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:

Example 1.2. A join query with a predicate.

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

Result:

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

1.1.2. Grouping Queries for JSON[1]

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"
: 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.

Example 1.3. A grouping query

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

Result:

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

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

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:

Example 1.4. A nesting grouping 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 (some $s in $store
                satisfies $sales."store number" eq $s."store number")
            and (some $p in $product
                satisfies $sales.product eq $p.name)
          let $pname := $sales.product
          group by $pname
          return { $pname : sum( $sales.quantity ) }
        |}
      }
    |}
  }
|}
```

Result:

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

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.

Collection "satellites":

```
{
  "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.

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

Example 1.5. A summarizing query.

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

Result:

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

1.1.4. 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 { "status" : "credit card declined" } into $dbj
```

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"
}
```

1.1.5. 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 := collection("youtube")
return
   copy $out := $feed
   modify
      let $feed := $out.feed
      let $feed-entry := $feed.entry
      for $entry at $pos in $feed-entry[]
      where $entry."app$control"."yt$state".name eq "restricted"
      return delete $feed-entry.pos
   return $out
```

Appendix A. Revision History

Revision 1.0.9 Mon Jul 22 2013 — **Ghislain Fourny** *ghislain.fourny@28msec.com*

Fixed new array unboxing syntax.

Revision 1.0.1 fri Apr 5 2013 — **Ghislain Fourny** *ghislain.fourny@28msec.com*

Ran all core queries on 28.io.

Revision 1.0.0 Thu Apr 4 2013 — **Ghislain Fourny** *ghislain.fourny@28msec.com*

First publication

Index