BRINGING NOSQL DATASTORES INTO AN XQUERY PLAYGROUND

Juan Zacarias @juanza_carias

Matthias Brantner

Cezar Andrei

A BRIEF INTRODUCTION

IT ALL BEGAN WITH THE NOSQL MOVEMENT

The name "NoSQL" movement (named after "Not Only SQL") was given to the flurry of activities in the area of distributed data storage and processing.

The movement's **main goals**:

- Being able to process **flexible schema data** (JSON or XML)
- Provide **scalability** for data processing

OUR FOCUS

In our paper and in this presentation we focus our attention on three major key-value data stores:

- **Mongo DB**
- **Couchbase**
- **Oracle NoSQL DB**

These data stores, allow the storage and access of binary data, or **JSON** in particular.

THE PROBLEM

JSON data stores **lack a structured query language** or any query language at all.

And when they do have query language, it is usually very limited in functionality. Often just able to search by key and simple inserts and replace updates.

The lack of a good query language is a major limitation in processing large amounts of JSON data productively and effectively.

A SOLUTION

We remember that:

- XML databases have **XQuery** as a sophisticated processing language.

Therefore, **why not JSONiq?** (JSONiq = XQuery + JSON)

This way we exploit decades of research, standardization, and implementation work around processing flexible data.

note: Querying JSON data is also the primary focus for the W3C Working group for the XQuery 3.1 extension.

To target this problem we proposed **three JSONiq modules** to interact, extract, query and update JSON data on the three NoSQL stores that were previously mentioned.

This way bringing the NoSQL databases into an existing XQuery playground , namely **Zorba**.

Since JSONiq is capable of querying both XML and JSON seamlessly, this three APIs allow the user to mix and match data extracted from NoSQL stores together with data stored in XML databases, effectively bringing **JSON query processing as par with XML**.

JSONIQ

JSONiq is a small and simple set of extensions to XQuery that add support to JSON.

Exploits the Fundamentals of XQuery

- The powerful FLWOR construct.
- The functional nature of the language.
- The modules.
- The declarative data updates.
- Full text search.
- Type systems, using a complete atomic type set.

In order to be able to also query JSON data, **JSONiq** add the following **extensions to XQuery**.

- Extensions to the XQuery Data Model (XDM) to support JSON (i.e. **objects and arrays**).
- Support for JSON datatypes (null or boolean), with a mapping to equivalent XML Schema types.
- **Navigation** for JSON Objects and JSON Arrays.
- **Constructors** for JSON Objects and JSON Arrays using the same syntax as JSON.
- **Update primitives** against JSON items as well as updating expressions that produce them.
- New **item types** that extend sequence type matching.

EXAMPLE DATASET & QUERIES

For purpose of **demonstrating the JSONiq language and the API's developed** we will be using a sample data set that contains information about the location, population, city and state for every zip code in the United States.

The data is given as a collection of 29469 JSON objects.


```
{
  "_id" : "10280",
  "city" : "NEW YORK"
  "pop" : 5547,
  "loc": [ -74.016323, 40.710537 ]
}
```

Querying

```
for $zip in jn:parse-json(file:read-text("zips.json"))
let $state := $zip("state ")
group by $state
let $pop := sum($zip("pop"))
where $pop > 10000000
order by $pop descending
return
{
  "state" : $state,
  "population" : sum($pop)
}
```

Updating

```
variable $object :=jn:parse-json(file:read-text("zips.json"))[1];
rename json $object("_id") as "zip_code";
delete json $object("loc");
$object
```

MONGODB

- A scalable, high-performace, open source NoSQL database.
- Stores and Query collections of JSON documents.
- Uses BSON (Binary JSON) which besides JSON primitive types supports some others, ex. byte, int, double, binary, datetime or timestamp

MongoDB's own query language works by creating "query documents" that indicate the patterns of the keys and values that are to be matched in the query.

JSONIQ API

The JSONiq module **exposes the most common** MongoDB operations.

- mongo:**connect**($options as object()) as xs:anyURI
- mongo:**collection-names**($db as xs:anyURI) as xs:string*
- mongo:**find**($db as xs:anyURI, $coll as xs:string) as object()*
- mongo:**find**($db as xs:anyURI, $coll as xs:string, $query as object(), $options as object(), $projection as object()) as object()*
- mongo:**save**($db as xs:anyURI, $coll as xs:string, $doc as object()) as empty-sequence()
- mongo:**update**($db as xs:anyURI, $coll as xs:string, $query as object(), $update as object()) as empty-sequence()
- mongo:**remove**($db as xs:anyURI, $coll as xs:string, $remove as object()) as empty-sequence()

CONNECTING

Connecting to the database is **as simple as passing connection information** (eg. host and database) as a JSON object to the connection function.

The connection function will return an opaque identifier (as xs:anyURI) to the database. This identifier is used by all the other functions as their first parameter.

```
let $db := m:connect ( { " host" : "localhost", "port" : 27017, "db" : "my-db" })
return (: do something with the connection $db :)
```

STORING DATA

The **save function** could be used to load data from a text file into MongoDB.

```
for $zip in jn:parse-json(file:read-text("zips.json"))
return m:save($db, "zips", $zip)
```

MongoDB uses BSON to store data. Since BSON extends the JSON data model with some other primitive types, the module defines a mapping from BSON types to types of the JSONiq data model(JDM).

Such mapping allows for round-trip documents without losing the types of the data.

ACCESSING DATA

In order to read data from a collection the module's **find function** can be used.

- The two argument version returns all the objects of the given collection.
- The five argument version allows for filtering and projecting the resulting documents.

```
for $zip in m:find($db, "zips")
let $state := $zip("state")
return (: do something with the data :)
```

IMPLEMENTATION NOTES

The implementation of the MongoDB module API has been done in C++ based on Zorba's API for building external modules.

It uses MongoDB's C++ driver for the communication with MongoDB.

The implementation of all the functions is relatively straightforward and has aprox 1300 lines of code.

However, this not include the mapping from JDM to BSON items and vice-versa. The mapping it is relatively complex because there is no obvious one-to-one mapping between those types. Designing and implementing this part was the hardest part of the module and it took another 3100 lines of code.

COUCHBASE SERVER

- Distributed key-value database.
- Efficient storing and retrieving arbitrary key-value pairs.

Couchbase has its own query language called views.

- Extract particular fields and information.
- Produces an index of the information.
- Goes beyond basic key-value retrieval (iterating, selecting and querying).
- Storing and Querying JSON.

JSONIQ API

The JSONiq module provides a way to **interact with Couchbase Server**.

- cb:**connect**($options as object()) as xs:anyURI
- cb:**get-text**($db as xs:anyURI, $key as xs:string*) as xs:string*
- cb:**get-binary**($db as xs:anyURI, $key as xs:string*) as xs:base64Binary*
- cb:**put-text**($db as xs:anyURI, $key as xs:string*, $value as xs:string*) as empty-sequence()
- cb:**put-binary**($db as xs:anyURI, $key as xs:string*, $value as xs:base64Binary*) as empty-sequence()
- cb:**remove**($db as xs:anyURI, $key as xs:string*) as empty-sequence()
- cb:**create-view**($db as xs:anyURI, $doc-name as xs:string, $view-name as xs:string*) as xs:string*
- cb:**create-view**($db as xs:anyURI, $doc-name as xs:string, $view-name as xs:string*, $options as object()*) as xs:string*
- cb:**view**($db as xs:anyURI, $path as xs:string*) as object()*
- cb:**view**($db as xs:anyURI, $path as xs:string*, $options as object()) as object()*

CONNECTING

The connect functions can be used to connect to the database, **as simple as passing connection information** (eg. host and bucket) as JSON object to the function.

As in the MongoDB module it returns a unique identifier (as xs:anyURI) which will be used in all the functions.

```
let $db :=cb:connect({ "host" : "localhost", "username" : "scott", "
password" : "tiger", " bucket" : "my-bucket" })

return (: do something with the connection $db :)
```

STORING DATA

The **put functions** can be used for getting data into Couchbase.

The JSONiq module API stores the data in two ways: as text or as binary data.

If we want to store JSON data we have to serialize it first.

```
for $zip in jn:parse-json(file:get-text("zips.json"))
return cb:put-text($db,  $zip("_id"), fn:serialize($zip))
```

ACCESSING DATA

The simplest way to access data is by using the **get functions**, ie. get-text and get-binary, they return the value for a given key as xs:string and xs:base64Binary respectively for each function.

```
let $data := cb:get-text($db, "id_number")
return (: do something with the data :)
```

Additional to the basic get functions this API was designed to **allow creating and querying views**. This is particularly important for JSONiq, because Couchbase has no other way for listing all or a subset of key-value pairs.

A **view** can be created using the create-view function. Essentially it allows the user to create a map using simple options.

```
cb:create-view ($db, "document-name", "view-name", { "key" : "doc.state" , "values" : "doc.pop" })
```

The view function can be used to return the content of the view given the name returned by create-view.

```
cb:view($db, $view-name)("rows")
```

Resulting in a sequence of JSON objects as follow:

```
{ _id = "94303", key = "CA", value="24309" }
```

If multiple values were specified in the definition, the values would be returned as an array.

```
let $view-name := cb:create-view(($db, "document-name", "view-name", { "key" : "doc.state" , "values" : "doc.pop" }))
for $zip in cb:view($db, $view-name)("rows")
let $state := $zip("state")
return (: do something with the data :)
```

IMPLEMENTATION NOTES

The implementation of this JSONiq module, just like MongoDB module, was done using the Zorba's API for external modules together with Couchbase server's C++ API.

The implementation of the module was a **straightforward communication** between zorba and Couchbase API's.

However some changes were needed to make the views available within JSONiq. These changes were mainly for making modifiable XQuery scripting to be translated into understandable javascript for the Couchbase Server.

ORACLE NOSQL DATABASE

- Distributed key-value database
- Built upon the Oracle Berkeley DB Java Edition high-availability storage engine.
- Provides highly reliable, scalable, and available data storage across a configurable set of systems that function as storage nodes.

Oracle NoSQL doesn't have a query language, it uses a composite key of multiple strings in which it can specify only the prefix of a key or a range of keys.

Giving more functionality to the user without the need of an actual query language.

JSONIQ API

The JSONiq modules defines the most common key value **store functions**:

- nosql:**connect**($options as object()) as xs:anyURI
- nosql:**get**($db as xs:anyURI, $key as xs:string) as xs:string
- nosql:**put**($db as xs:anyURI, $key as xs:string, $value as xs:string) as xs:long
- nosql:**delete**($db as xs:anyURI, $key as xs:string) as xs:boolean
- nosql:**multi-get-base64**($db as xs:anyURI, $parentKey as object(), $subRange as object(), depth as xs:string, $direction as xs:string) as object()*
- nosql:**multi-get-string**($db as xs:anyURI, $parentKey as object(), $subRange as object(), $depth as xs:string, $direction as xs:string) as object()*
- nosql:**multi-get-json**($db as xs:anyURI, $parentKey as object(), $subRange as object(), $depth as xs:string, $direction as xs:string) as object()*
- nosql:**multi-delete-values**($db as xs:anyURI, $parentKey as object(), $subRange as object(), $depth as xs:string) as xs:int

CONNECTING

As the previous modules, the **connect function** takes the address and name of the database as argument and returns a connection identifier.

This identifier is used by all the functions to discriminate between multiple connections.

```
let $db := nosql:connect({
  "store-name" : "kvstore",
  "helper-host-ports" : ["localhost:5000"] }
)
return (: do something with the connection $db :)
```

STORING DATA

Since Oracle NoSQL DB is a composed key-value store, we use the put functions to store data. In the example we chose to load the data having the major key as 'zips' and under minor key as the zip-code '_id'.

```
for $zip in jn:parse-json( file:read-text("zips.json") )
return
  nosql:put-json($db,
                 { "major" : ["zips"],
                   "minor": [{$zip("_id")}] },
                 $zip )
```

ACCESSING DATA

As for any key-value store, the simplest way to access data is using the get function. In Oracle NoSQL database there are more advanced functions that allow to return multiple objects at once.

```
for $zip in nosql:multi-get-json($db,
                                 { "major" : "zips" }, {"prefix": ""},
                                 $nosql:depth-PARENT_AND_DESCENDANTS,
                                $nosql:direction-FORWARD)(“value”)
let $state := $zip("state")
return (: do something with the data :)
```

IMPLEMENTATION NOTES

Oracle NoSQL module is implemented using NoSQL's Java client library. The module covers the most important aspects of the NoSQL API, leaving out many details for corner cases.

This makes the module simple to use while powerful with access to more advanced compound keys and multi get/delete functionalities

CONCLUSIONS & OUTLOOK

This presentation showed us ***three JSONiq modules*** each which allows interfacing with a NoSQL database.

The goal of these modules is to allow NoSQL developers to use the full power of JSONiq with data in their favorite NoSQL database.

In the future we would like to define modules to the other NoSQL stores (e.g. Amazon, DynamoDB).

We would also like to share our findings with the W3C working group in order to officially integrate JSON data processing into XQuery.

In the meantime we hope that the proposed connectors help NoSQL database vendors to get a feeling of using a sophisticated data query language..

The expressive power of JSONiq/XQuery together with the scalability of NoSQL datastores make a killer combination.

THANKS

REFERENCES

- MongoDB Module http://my.zorba-xquery.com/tmp/XMLPrague2013/mongodb.xq and http://my.zorba-xquery.com/tmp/XMLPrague2013/mongodb.xsd.xml
- Couchbase Server Module http://my.zorba-xquery.com/tmp/XMLPrague2013/couchbase.xq
- Oracle NoSQL DB Module http://my.zorba-xquery.com/tmp/XMLPrague2013/nosqldb.xq
- JSONiq Language Specification http://www.jsonig.org/
- MongoDB http://www.mongodb.org/
- Couchbase Server http://www.couchbase.org/
- Oracle NoSQL DB http://www.oracle.com/technetwork/products/nosqldb/overview/index.html
- Zorba http://www.zorba-xquery.org/