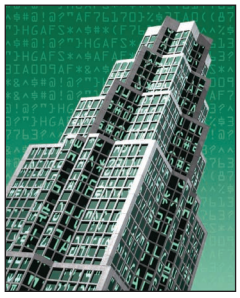
Editor: Tim Kraska • kraskat@cs.brown.edu



JSONiq: The History of a Query Language

Daniela Florescu • *Oracle*

Ghislain Fourny • *28msec*

NoSQL data stores support hierarchical, heterogeneous data such as XML, JavaScript Object Notation (JSON), and Binary JSON (BSON). A need exists for a tailor-made language for querying such hierarchical, heterogeneous datasets. JSONiq was designed from the ground up to support JSON data and run against all JSON- or BSON-based data stores, turning them into full-fledged databases.

Edgar F. Codd was initially trained as a mathematician at the Exeter College, Oxford. Unsurprisingly, then, his major contribution to computer science — the relational data model he proposed in 1970[1] — had the simplicity and elegance of mathematical abstractions. For the next 40 years, the relational model became the primary way of representing the forever-increasing volume of data that we store. Codd's relational data model gave birth to a major scientific field and a multibillion dollar industry. We can't imagine modern civilization without the relational data model, or without SQL, the declarative query language proposed in the mid-1970s by Don Chamberlin and Raymond Boyce.[2] SQL was designed to query and process data modeled as relations, and proved to have the properties that people needed: it was productive, and it was open for automatic optimization.

Unfortunately, as early as 1990, evidence was growing that the world of information that we need to model, store, and process isn't simple, elegant, or very symmetrical. Information is messy, yet messy information has a lot of value. Two turning points occurred between the birth of relational databases and now.

The Relational Era

The first moment was triggered, ironically, by database technology's success. The more data we had, the more applications we wrote to use it. Unfortunately, application code was written in nondeclarative programming languages such as C++ or, later, Java, which didn't use the relational model to represent the data. Moving data back and forth between SQL and these application languages created friction, both at the productivity and performance levels. In the nineties, this problem was referred to as the *impedance mismatch*. The same problem was also referred to as *the arm struggle between applications and databases*, and it left the IT industry in a very unhealthy state.

So, why does the impedance problem occur? It's simple: Codd's model sees data as sets of flat records (the famous *first normal form*), while applications see the same data as complex structures with nested substructures. Consequently, the code on both sides of the fence doesn't match, and expensive operations are necessary to move data between databases and applications. During the 1990s, object-oriented databases attempted to solve this problem. They failed to gain economical success, yet the Object-Oriented Query Language (OQL), a functional language that incorporated the main ideas behind SQL, had a significant impact on the minds of young database researchers. It was more modern, more symmetrical, and simpler than SQL, yet maintained SQL's properties: it was productive and optimizable.

The Web Era

The second wave of change was triggered by the advent of the Web. Around the mid-1990s, more and more data was exposed on the Web. This data had to be captured, stored, and processed.

Published by the IEEE Computer Society

1089-7801/13/$31.00 © 2013 IEEE

However, it differed from the business data that Codd had in mind in 1970. First, it wasn't homogeneous. Data was still modeled as sets of records, yet the records weren't all alike: not all companies on the Web have the same attributes, nor do all people on the Web have the same attributes, nor are all purchase orders alike. Data is heterogeneous. The second problem was that Web data was evolving fast, much faster than before. The old strategy of having a fixed schema that rarely changed wasn't working. Soon, schemas became irrelevant, and data had to be freed from schema constraints, or else lots of it would fail to be captured, stored, and processed.

XML was proposed in 1996 to solve several of these problems: it was a solution for complex nested structures, as well as schema-less data. XML was a simplification of the Standard Generalized Markup Language (SGML), an old model used to represent documents — that is, human-generated information. For a time, many database practitioners saw XML as a solution for their problems. This soon led to disappointment: XML was too complicated; it included many details from SGML that seemed irrelevant to modeling data and badly affected both productivity and performance with regard to accessing data. For the next decade and a half, however, XML was the only option for schema-flexible data with complex structure. So, lots of IT professionals used it by necessity, and will probably continue to do so.

XML technology slowly started to mature, with a (flexible and open!) schema language (XSD), a transformation language (XSLT), and a declarative query language (XQuery). The latter was built on 40 years of experience with SQL and OQL: no coincidence, then, that its authors had deep roots in relational database and object-oriented technology. XQuery is a functional query language that lets users freely combine a small set of well-designed, complementary expressions: arithmetics, comparisons, conditionals, navigation, and — last but not least — the FLWOR expression.

The FLWOR expression is a generalization of SQL's famous `SELECT-FROM-WHERE`. In a single, simple, and compact expression, we can express selections, projections, joins, aggregates, sorts, streaming queries, and storing of intermediate results during a complex query. The theoretical models behind FLWOR expressions are mathematical *monoid comprehensions*. Such expressions are the core of the XQuery language and provide an excellent basis for rewrites and query optimization, similar in nature to the optimizations performed on SQL queries, but also allowing many optimizations borrowed from traditional compiler techniques.

XQuery's expressiveness created a large industry: most traditional database and middle tier vendors (such as IBM, Microsoft, and Oracle) support it in their existing infrastructures, while other vendors (MarkLogic and 28msec, for example), and open source projects (such as Saxon, eXist, BaseX, or Zorba) focus entirely on XML and XQuery.

Enter JSON and JSONiq

Despite the wide use of XML and XQuery, database practitioners still considered them suboptimal for modeling and processing semi-structured data. In 2001, Douglas Crockford proposed a much simpler variant of XML: JavaScript Object Notation (JSON).[3] Like XML, and unlike Codd's relational model, JSON is a serialization format, not a logical data model. Like XML, it can model records with complex, nested structure, and schema-less data. Yet, unlike XML, it can model only records, not human-generated documents, so that parsing JSON is much simpler and more efficient.

On the Shoulders of Two Giants

Two years ago, we started a project to introduce a logical data model for JSON (the JSON Data Model, or JDM) and to adapt XQuery to it. The recent outcome is the JSONiq query language, built not as a brand new query language, but rather on the shoulders of giants: SQL and XQuery.

JSONiq borrowed several ideas from XQuery, such as the structure and semantics of a FLWOR construct, the language's functional aspect, the semantics of comparisons in the face of data heterogeneity, and declarative, snapshot-based updates. However, unlike XQuery, JSON isn't concerned with the peculiarities of XML, such as mixed content, ordered children, the confusion between attributes and elements, the complexities of namespaces and QNames, or the complexities of XML Schema. The power of XQuery's FLWOR construct and the functional aspect, combined with the simplicity of the JSON data model, result in a clean, sleek, and easy to understand data processing language. In fact, JSONiq is a language that can do more than queries: it can describe powerful data processing programs, including transformations, selections, joins of heterogeneous datasets, data enrichment, information extraction, information cleaning, and so on.

Properties of the JSONiq Language

Technically, JSONiq (and XQuery) has the following main characteristics.

First, it's *set-oriented*. While most programming languages are designed to manipulate one object at a time, JSONiq is designed to process sets (actually, sequences) of data objects.

Second, it's *functional*. A JSONiq program is an expression; the program's result is the result of the expression's evaluation. Expressions

have a fundamental role in the language: every language construct is an expression, and expressions are fully composable.

Third, it's *declarative*. A program specifies what result is being calculated and doesn't specify low-level algorithms such as the sort algorithm, or the fact that an algorithm is executed in main memory or is external, is on a single machine or parallelized on several machines, or what access patterns (that is, indexes) are in use during the program's evaluation. Such implementation decisions should be made automatically by an optimizer based on the physical characteristics of the data and the hardware environment, as with a traditional database. We designed the language from day one with optimizability in mind.

Fourth, it's *designed for nested, heterogeneous, semi-structured data*. Data structures in JSON can be nested with arbitrary depth, don't have a specific type pattern (that is, they're heterogeneous), and can have one or more schemas that describe the data. If a schema exists, it can be open, or can simply partially describe the data. Unlike SQL, which is designed to query tabular, flat, homogeneous structures, JSONiq is a tailor-made query language for nested and heterogeneous data. In particular, it has the following characteristics that are necessary for processing irregular, schema-less data:

- The semantics of basic operators (such as comparisons, arithmetics, and aggregates) are designed to be robust to variations in the data types — unlike the traditional query languages, which expect all records to be similar in structure.
- It provides a solid error-management capability that blends uniformly with the rest of the language, which is necessary for dealing with unexpected data.
- It includes recursion as a built-in mechanism to process data with deep, arbitrarily complex structure.
- It provides simple, built-in operators for navigating through arrays and objects.

In the relatively short time since its inception, JSONiq has received considerable attention. Several database (for example, 28msec) and middle-tier software applications (Zorba and IBM's Web Sphere) have implemented it.

A Short Introduction to JSONiq

Here, we look at JSONiq's basic building blocks and describe several data processing scenarios that can be expressed using the JSONiq language.

JSON as a Subset of JSONiq

JSONiq was tailor-made for the JSON syntax. Consequently, any JSON building block is also a valid JSONiq query. For example, this is a JSON document (taken from a JSON dataset of Stack Overflow FAQs), and like all JSON documents, it's a valid JSONiq query:

```
{
 "_id" :
"511C7C5C9A277C22D138802D",
 "question_id" : 4419499,
 "last_edit_date" : "2012-12-
17T00:02:31",
 "score" : 15,
 "title" : "MySQL and NoSQL:
Help me to choose the right
one",
 "tags" : [ "php", "mysql",
"nosql", "cassandra" ],
 "owner" : {
   "user_id" : 279538,
   "display_name" : "cedivad",
 }
}
```

The JSONiq Data Model

From a data model perspective, JSONiq supports the following so-called items:

- *Objects* — unordered sets of key-value pairs, with no duplicate keys. Keys are atomics (strings), and values can be any item.
- *Arrays* — ordered lists of values. A value can be any item.
- *Atomics* — leaf values annotated with types. Atomic types involve JSON counterparts, such as string, integer, decimal, double, boolean, and null; further useful built-in types support date and time, binary formats such as base64, and so on.

As mentioned, JSONiq is set-oriented, meaning that expressions take and return sequences of items. Sequences are flat, and a sequence of one item is considered to be the same as that item.

Composing Expressions

Starting with the previously mentioned JSONiq subset (the JSON syntax), we can compose expressions with a high degree of freedom. JSONiq supports basic operations such as generating an arithmetic sequence, arithmetics, or string concatenation:

```
[ 1 to 5 ]
→ [ 1, 2, 3, 4, 5 ]
{ "two" : 1 + 1 }
→ { "two" : 2 }
"foo" || "bar"
→ "foobar"
```

Navigation is done with dots for object lookup, and with `[[ ]]` for array lookup — for example, `collection("faqs").tags[[3]]`. It's also done with an implicit iteration — that is, `$objects.foo` returns the values associated with the key "foo" in all the input sequence's objects (non-objects are ignored without raising any error). `$arrays[[3]]` returns the values that are in the third position of each array in the input sequence (likewise, non-arrays are ignored).

JSONiq supports comparison operators as well as two-valued logic, which can be used in conditional expressions. For convenience, many non-boolean atomic values are converted automatically to booleans in an intuitive way.

For example, the empty string gets converted to false and the string "foo" to true. Zero is converted to false, while non-zero numbers are converted to true. This behavior is similar to that of C++ in spirit.

FLWOR Expressions

The most powerful expression in JSONiq is the FLWOR expression, which generalizes its SQL counterpart. You can join, project, select, group, filter, aggregate, and so on.

Figure 1 (i) iterates with a *for* clause on objects in a collection of answers; (ii) accesses with a *let* clause the IDs of the question that they answer; (iii) groups with a *group by* clause the answers by question IDs (answers to the same question will be together); (iv) filters with a *where* clause those groups that have more than one answer; and finally (v) outputs with a *return* clause aggregated information for each group — the question ID and average answer score.

Note that parentheses must be used in certain places to override precedence.

A Rich Function Library

JSONiq comes with a rich function library involving aggregation functions (such as *count*, *avg*, *sum*, *max*, *min*), JSON functions (*keys*, *members*, *parse-json*, *size*, *serialize*), functions on strings (*substring*, *concat*, *string-join*, *upper-case*, *lower-case*, *tokenize*), functions on sequences (*subsequence*, *insert-before*, *remove*, *deep-equal*, *distinct-values*), functions on numbers (*ceiling*, *floor*, *round*, *abs*), miscellaneous other functions (*current-dateTime*, *error*, *boolean*), and a math library (with functions such as *pow*, *exp*, *exp10*, *log*, and *sin*).

Miscellaneous

JSONiq is a strongly typed language, but types aren't in the programmer's way. For the previous query, variables with no type indication are assumed to have the topmost static type item*.

```
(i)   for $answer in collection("answers")
(ii)  let $qid := $answer.question_id
(iii) group by $qid
(iv)  where count($answer) gt 1
(v)   return {
        "question" : $qid,
        "count" : avg($answer.score)
      }
```

Figure 1. FLWOR expression. Such expressions let you join, project, select, group, filter, aggregate, and so on.

```
      import module namespace file = "http://expath.org/ns/file";
(i)   let $text := file:read-text("faq.json")
(ii)  let $faq-object := parse-json($text)
(iii) let $new := {
        "Titel" : $faq-object.title,
        "Author" :
            $faq-object.owner.display_name,
        "Note" : $faq-object.score,
        "Entstanden" :
            date($faq-object.creation_date)
      }
(iv)  return insert("faqs", $new)
```

Figure 2. Loading a document. The query reads and parses a JSON object from a file, then transforms it to a new format.

```
(i)   for $answer in collection("answers")
(ii)  group by $qid := $answer.question_id
(iii) return {
      "question" : collection("faqs")[$$.question_id eq $qid].title,
      "average score" : avg($answer.score),
      "max score" : max($answer.score),
      "duplicate posters" : [
        for $a in $answer
        group by $name := $a.owner.display_name
        where count($a) gt 1
        return $name
      ]
    }
```

Figure 3. Performing analytics with JSONiq. This query computes statistics on Stack Overflow answers, grouping them according to the question to which they relate. For each question, it outputs that question's formulation, the average and max answer score, and a list of people who answered twice.

Many more expressions are available, such as *switch*, *typeswitch*, *try-catch*, *cast*, *instance-of*, *treat*, or first-order-logic quantifiers. JSONiq supports user-defined functions and user-defined library modules as well.

```
    import module namespace full-text = "http://www.zorba-xquery.com/
    modules/full-text";
(i) for $faq in collection("faqs")
(ii) for $synonym in full-text:thesaurus-lookup("Data")
(iii) where contains($faq.title, $synonym)
(iv) return $faq
```

Figure 4. Full-text search. This query looks for synonyms of a text query ("Data").

```
    [
        distinct-values(
          for $answer in collection("answers")
          let $oid := $answer.owner.user_id
(iii)     where count(
            for $question in collection("faqs")
            where
(v)         some $other-answer
            in collection("answers")
              [$$.question_id eq
                   $question.question_id
               and
               $$.owner.user_id eq $oid]
            satisfies
(vi)         $other-answer.score gt $answer.score
            return $question
(iv)      ) ge 2
(ii)      where not $answer.is_accepted
(i)       return $answer.owner.display_name
        )
    ]
```

Figure 5. Advanced, nested query. This query looks for users who got an answer not accepted, but for whom there were at least two questions for which they gave an answer with a better score.

Applications

Now that we've introduced the basics of the language, let's look at three typical application examples.

Load a Document

The query in Figure 2 (i) reads and (ii) parses a JSON object from a file. It then (iii) transforms it to a new format with German keys and converts the creation date to an atomic of type *date*. Finally, (iv) the new object is inserted into the existing *faqs* collection. The latter operation uses the insert function from JSONiq's update capability.

Analytics

JSONiq lets you perform analytics on your data. The query in Figure 3 computes statistics on (i) Stack Overflow answers. It (ii) groups the answers by the question to which they relate, and then (iii) outputs, for each question, the question's formulation, the average and max answer score, and the list of people who answered twice.

Full-Text with Streaming

Finally, JSONiq also supports full-text search. The query in Figure 4 looks for synonyms of a text query ("Data"). It (i) iterates over frequently asked questions, (ii) iterates over synonyms of a full-text query ("data"), (iii) filters the successful matches, and (iv) returns them.

A More Advanced Example

Figure 5 shows an example of a more advanced, nested query. It looks for (i) users who (ii) got an answer not accepted, (iii) but for whom there were (iv) at least two questions for which (v) they gave an answer with (vi) a better score.

JSONiq is already available in stable releases — for example, in the Zorba NoSQL (in-memory) processor (http://zorba.io/), and on top of the 28.io platform (turning MongoDB data stores into databases; see http://28.io/). JSONiq is available under a license that makes it free to copy, distribute, share, and implement, with the goal of interoperability between various NoSQL data stores. ◻

Acknowledgments

JSONiq is a team effort involving Jonathan Robie, Matthias Brantner, Till Westmann, Markos Zaharioudakis, and ourselves.

References

1. E.F. Codd, "A Relational Model of Data for Large Shared Data Banks," *Comm. ACM*, vol. 13, no. 6, 1970, pp. 377–387.
2. D. Chamberlin and R. Boyce, "SEQUEL: A Structured English Query Language," *Proc. ACM SIGFIDET Conf.*, ACM, 1974.
3. D. Crockford, *The Application/JSON Media Type for JavaScript Object Notation (JSON)*, IETF RFC 4627, July 2006; www.ietf.org/rfc/rfc4627.txt.

Daniela Florescu is a NoSQL software architect at Oracle. Her research interests include NoSQL databases and Web services. Florescu has a PhD in computer science from the University of Paris 6. Contact her at daniela.florescu@oracle.com.

Ghislain Fourny is a software architect at 28msec. His research interests include languages for querying and updating NoSQL databases as well as parallelism. Fourny has a PhD in science from the Swiss Federal Institute of Technology (ETH Zürich). Contact him at ghislain.fourny@28msec.com.