Requesting Heterogeneous Data Sources with Array Comprehensions in Hop.js

Yoann Couillec Manuel Serrano
Inria / Université Côte d'Azur, France
{Yoann.Couillec,Manuel.Serrano}@inria.fr

Abstract

During the past few years the volume of accumulated data has increased dramatically. New kinds of data stores have emerged as NoSQL family stores. Many modern applications now collect, analyze, and produce data from several heterogeneous sources. However implementing such applications is still difficult because of lack of appropriate tools and formalisms. We propose a solution to this problem in the context of the JavaScript programming language by extending array comprehensions. Our extension allows programmers to query data from usual stores, such as SQL databases, NoSQL databases, Semantic Web data repositories, Web pages, or even custom user defined data structures. The extension has been implemented in the Hop.js system. It is the subject of this paper.

Categories and Subject Descriptors H2.3 [*Languages*]: Query Languages; H3.3 [*Information Search and Retrieval*]: Query formulation

Keywords JavaScript; array comprehension; language-integrated query; database; Web page; Web service; compilation; aggregation.

1. Introduction

Creating applications involving data sources raises several programming problems. First, integrating a query language within a programming language suffers the well-known impedance mismatch problem [1]. Second, as the data sources rely on different models, they are generally interrogated with specific means such as a dedicated query languages, ad-hoc APIs, or sets of HTTP requests. This creates a second impedance mismatch between the sources themselves. Our contribution proposes a unique formalism to query over multiple sources. It is based on JavaScript array comprehensions.

Merging data query languages and algorithmic programming languages is an old problem. Solutions have been long proposed and are now widely deployed. For instance, the popular Java Hibernate framework enables Java programs to conveniently access databases [12] or the Links [6] programming language, merges the algorithmic language and the database language inside a single programming language.

Modern applications use different kinds of data: *i)* raw data, which are used to encode unorganized information like media resources, *ii)* structured data, which are found in relational databases [11], and *iii)* data streams which are, for instance, produced by censors. This heterogeneity raises programming problems for applications that combine multiple data sources. Linq [2, 3] and F#'s Type Providers [5] address this problem. They allow the programmers to write queries on arbitrary data sets such as collections, XML data, and relational databases. Linq demonstrated that comprehensions constitute a convenient means for expressing queries since data are collections of objects.

Links solves the impedance mismatch problem by combining the code of the three tiers: the server, the client, and the database. Links compiles the queries to SQL [17] and XQuery [14].

In a recent work [4], Cheney, Lindley, and Wadler combine Linq and Links in a single language named PLinq. It employs quoted expressions to queries over relational databases. It relies on normalization of quoted terms. The authors have proved that one query over a unique database produces a unique SQL query [13] and that the PLinq query normalization process always succeeds. We reuse this normalization process in our work.

In this paper, we show how to use array comprehensions for querying over several data sources evenly. We based our solution on Links and PLinq for merging the query language and the programming language and on Linq for supporting multiple backends. Our extension to JavaScript array comprehension has been implemented in Hop.js, a multitier extension of JavaScript. It offers the possibility of requesting multiple and heterogeneous data sources with a single query. It supports usual data stores, such as relational databases, NoSQL databases, etc. It also supports less traditional sources of data such as Web pages and Web services. Furthermore, it allows users to develop their own custom sources backends.

2. Array Comprehensions as a DSL for Querying Data Sources

Comprehensions are defined by the mathematics set theory [8]. In their original form, they are written as:

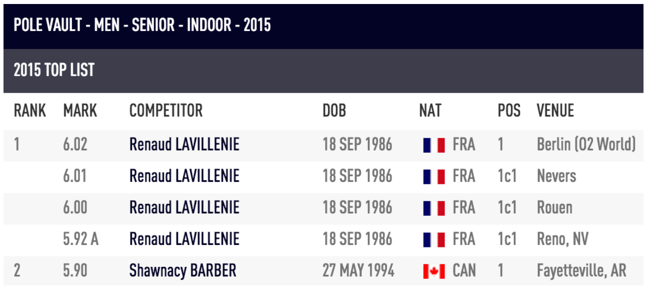
$$\{\ x^2 \mid x \in A \wedge odd\ x\ \}$$

In this example, the comprehension expresses the set of all square of x such as x belongs to the set A and such as x is odd. As argued before [9], this construction is a clear, concise, and efficient way to express queries. Trinder and Wadler [10] have shown that any relational calculus queries can be expressed as a comprehension.

In this section, we recap the main components of the ECMAScript 7 array comprehension proposal. Then, we present two examples that show how to query data sources with comprehensions.

comprehension := [*iterable+ filter? expression*]
iterable := `for` (*javascript-lhs* `of` *javascript*)
filter := `if` (*javascript*)
expression := *javascript*

Figure 1. Array comprehension syntax



POLE VAULT - MEN - SENIOR - INDOOR - 2015

2015 TOP LIST

RANK	MARK	COMPETITOR	DOB	NAT	POS	VENUE
1	6.02	Renaud LAVILLENIE	18 SEP 1986	FRA	1	Berlin (O2 World)
	6.01	Renaud LAVILLENIE	18 SEP 1986	FRA	1c1	Nevers
	6.00	Renaud LAVILLENIE	18 SEP 1986	FRA	1c1	Rouen
	5.92 A	Renaud LAVILLENIE	18 SEP 1986	FRA	1c1	Reno, NV
2	5.90	Shawnacy BARBER	27 MAY 1994	CAN	1	Fayetteville, AR

Figure 2. IAAF: A Web page as a data source (`http://www-.iaaf.org/records/toplists/pole-vault/indoor/men/senior/-2015`).

2.1 Syntax

Comprehensions are supported by many languages such as Haskell, F#, and Python. In JavaScript, comprehensions are supported in the version 1.7[1]. The syntax[2] is presented in Figure 1.

A comprehension is composed of *iterable objects*, an optional *filter* and an *expression*. Evaluating a comprehension produces a fresh array whose elements are obtained by evaluating the *expression* for every element of *iterable objects* that is accepted by the *filter*. Multiple sources can be used simultaneously. The non-terminal token *javascript-lhs* stands for *left-hand-side expression* [3]. Using ECMAScript comprehensions, the squares of odd numbers are expressed as:

```
[ for ( x of A ) if ( odd ( x ) ) x * x ]
```

If `A` is `[1,2,3,4]`, the result of the comprehensions is the array `[1,9]`.

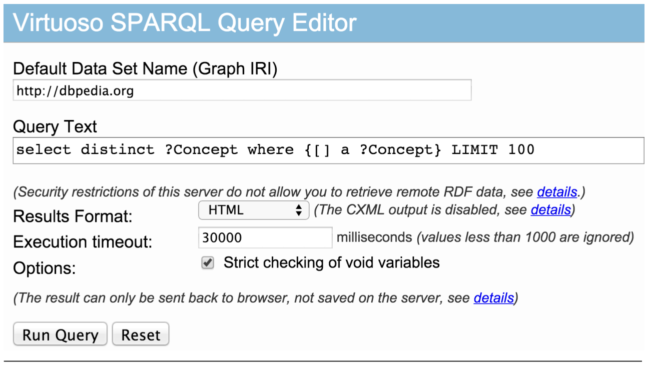
We will now show our generalization of comprehensions. We will show how to use them with other types of sources than JavaScript arrays. We will also show how our extension allows programs to query data from heterogeneous data sources.

2.2 Using Comprehensions

In this section, we show that array comprehensions can be used beyond JavaScript arrays. For the sake of the examples we show how to use them with Web pages, and SPARQL endpoints. We base our presentation on two practical test cases: IAAF and DBpedia.

2.2.1 Web Pages: IAAF

IAAF is a Web site that presents athletics competition results (see Figure 2 for the 2015 pole vault results). Each row contains an athlete's name, his mark for the race, the date, the event, and also visual additional information such as a country flag.



Virtuoso SPARQL Query Editor

Default Data Set Name (Graph IRI)
http://dbpedia.org

Query Text
select distinct ?Concept where {[] a ?Concept} LIMIT 100

(Security restrictions of this server do not allow you to retrieve remote RDF data, see details.)
Results Format: HTML (The CXML output is disabled, see details)
Execution timeout: 30000 milliseconds (values less than 1000 are ignored)
Options: ☑ Strict checking of void variables
(The result can only be sent back to browser, not saved on the server, see details)

Figure 3. DBpedia: The Semantic Web as a data source (`http://dbpedia.org/sparql`)

The IAAF pages are not designed to be processed by computer programs; they are designed for humans. However, with our array comprehension extension, they can be accessed programmatically, which lets the IAAF information be used differently. For instance, the following single request

```
[ for ( x of iaaf.pole_vault.2015 )
   if ( x.mark > 6 )
     { name: x.name, mark: x.mark } ]
```

extracts the athletes' names and their mark for all pole vault higher than 6 meters.

In this example `iaaf.pole_vault.2015` is the data source, which is queried using the comprehension. It encapsulates the URL that points to the 2015 pole vault results. The variable `x` is successively bound to all the objects that represent the IAAF results. The filter part of the comprehension selects only the objects whose `mark` is greater than 6. For each of these objects, a fresh JavaScript object containing the `name` of the athlete and the `mark` of the result is built. These objects are accumulated in the array that is the result of the evaluation of the comprehension. The IAAF data source is created as follows:

```
var Datasource = require ("./datasource.js");
var Iaaf = require ("./datasource-iaaf.js" );
var iaaf = Datasource.create (Iaaf);
iaaf.pole_vault.2015 =
 iaaf.createTable
      ("http://www.iaaf.org/records/toplists/
        pole-vault/outdoor/men/senior/2015");
```

This first imports the `datasource.js` library. Then, it creates a JavaScript object representing the IAAF Web site and it creates a data source corresponding to men's pole vault results of 2015.

2.2.2 SPARQL Endpoints: DBpedia

DBpedia is the Semantic Web version of Wikipedia. It uses an RDF-graph data model and SPARQL as the query language. DBpedia uses Virtuoso, a database that provides a Web interface (see Figure 3). It allows users to write and send SPARQL queries.

The following comprehension shows how to get the birth city of Barack Obama:

```
[ for ( p of dbpedia.person )
   for ( city of dbpedia.location )
     if ( p.name == "Barack Obama"
       && p.birthPlace == city)
       city.name ];
```

This comprehension uses two `for` statements as the request involves two data sets: `dbpedia.person` and `dbpedia.location`.

---

[1] `https://developer.mozilla.org/en-US/docs/Web/JavaScript/New_in_JavaScript/1.7`

[2] `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Array_comprehensions`

[3] `http://www.ecma-international.org/ecma-262/5.1/\#sec-11.2`

The evaluation of the comprehension joins the two data sets on the city name.

In this example, the variable `p` is bound to persons and the variable `city` is bound to cities. The filter performs the join operation and selects the person named “Barack Obama”. The `dbpedia.person` and `dbpedia.location` are built with:

```
var Datasource = require ("./datasource.js");
var DBpedia = require ("./datasource-dbpedia.js");

var dbpedia = Datasource.create
   (DBpedia ("http://dbpedia.org/sparql"));

dbpedia.person =
 dbpedia.createTable
   ("http://xmlns.com/foaf/0.1/Person",
     [{column: "name",
        uri: "http://xmlns.com/foaf/0.1/name"},
      {column: "birthDate",
       uri: "http://dbpedia.org/ontology/date"}]);
dbpedia.location =
 dbpedia.createTable
   ("http://dbpedia.org/ontology/Place",
     [{column: "name",
       uri: "http://xmlns.com/foaf/0.1/name"}]);
```

This code creates a data source, `dbpedia`, two tables `person` and `location` with their columns, which are used in the comprehension.

3. Array Comprehension Plugins

We present here the generic architecture we have designed to allow comprehensions to operate over data sources. This architecture is open as it allows user programs define their own comprehension backends by means of *plugins*. For the sake of simplicity, in this section we assume comprehensions over single sets.

As seen before, a comprehension is composed of three elements: an *iterable*, a *generator expression*, and a *filter expression*. The compilation of each of these elements depends on the nature of the data source. For instance, the DBpedia comprehension backend compiles the query into SPARQL while the IAAF backend compiles the query into a sequence of URL downloading and HTML parsing actions. Second, the backend compiles the *filter*. The very nature of this compilation highly depends on the capacities of the backend. In the least efficient case, this compilation generates a JavaScript predicate that is applied once all the records have been extracted from the data source. In the most efficient case, the filter is fully compiled into the query language of the data source when it supports one which is expressive enough. Third, the backend compiles the generator, mixing JavaScript code generation and the possibilities offered by the data source.

The appropriate plugin associated with a comprehension depends on the type of the data source, which is only known at runtime. Thus, a comprehension cannot be entirely compiled before the execution starts. Half of the compilation occurs during the static compilation of the program. During that stage, the comprehension is compiled as if it was a plain array comprehension. This compilation also inserts serialized forms of the abstract syntax trees (AST) representing the filter, the generator, and the variables to which data sources are bound. These AST are used at runtime by the second half of the comprehension compilation, which is specific to each plugin.

When a comprehension is to be evaluated, the appropriate plugin is selected by inspecting the dynamic type of the data source. Using the JavaScript late binding mechanism, the data source plugin implementation is then obtained. The plugin defines *i)* the query compiler that uses the AST produced by the static compilation, and *ii)* the runtime system that executed the query.

3.1 Generic Plugin Definition

An array comprehension plugin is characterized by five methods:

1. `predicate`: a predicate which is true if and only if the filter expression can handled natively by the data source. This predicate is applied on the filter AST. If the predicate is false, the filter will be implemented in JavaScript and used on all the values fetched from the data source.
2. `compileTable`: a function that compiles the iterable object into the formalism of the data source query language. It takes three arguments: the initial query, the variable identifier of the comprehension expression, and the iterable object.
3. `compileProjectionGenerator`: a function that compiles the projections from the generator expression. It takes three arguments: the generator AST, the initial query and information about the columns.
4. `compileFilter`: a function that compiles the filter. It takes two arguments: the initial query and the filter AST.
5. `execute`: a function that executes the native query.

In the following section we show an example of array comprehension plugins implementation.

3.2 Plugin Example

As an example, we now describe the implementation of the DBpedia plugin. In this example, only simple filters are compiled into SPARQL: the filter predicate is true only for expressions containing projections, literals, and `||` or `&&` binary operators. More complex predicates are handled in JavaScript. The filter might be implemented as:

```
predicate : function (astf) {
 function isValid (token) {
   return token == "&&"
       || token == "||"
 }
 function areValidChildren (children) {
   if (children.length == 0) {
     return true
   } else {
     var first = children.shift () ;
     var rest =  children ;
     return isValid (first.token)
         && areValidChildren (first.children)
         && areValidChildren (rest)
   }
 }
 return isValid (astf.token)
     && areValidChildren (astf.children);
}
```

The `compile`*XXX* functions construct the SPARQL request to be generated as an abstract syntax tree, which is represented by a JavaScript structure containing three fields: `select`, `where`, and `filter`. Each of the `compile`*XXX* functions takes an AST as argument and modifies it, generally by adding new elements. Once the SPARQL AST is fully built, it is then pretty printed into the concrete SPARQL syntax by the function `sparqlToString` function. The abstract representation is a JavaScript object looks like:

```
{ select : ["?x_name"],
 where : [ { subject: "?x",
```

```
             predicate: "a",
             object: "dbpedia-owl:Person" },
           { subject: "?x",
             predicate: "foaf:name",
             object: "?x_name" } ],
 filter : { type: "==", left: ... } }
```

This AST represents the following SPARQL query:

```
SELECT ?x_name {
 ?x a dbpedia-owl:Person .
 ?x foaf:name ?x_name .
 FILTER (?x_name == "Obama"@en)
}
```

Assuming the method `extractProjections` that extracts the projections of the generator from an AST, the `compileProjectionGenerator` method can be defined as:

```
compileProjectionGenerator:
 function (astgen, query, columns) {
   var projections = astgen.extractProjections ();
   function projectionToSelectString (proj) {
     return proj.obj + "_" + e.field
   }
   function projectionToWhere (proj) {
     return {
       subject: proj.obj,
       predicate: columns[proj.field],
       object: projectionToSelectString (proj)
     }
   }
   return {
     select: query.select.concat(
       projections.map(projectionToSelectString)),
     where: query.where.concat(
       projections.map(projectionToWhere)),
     filter: query.filter
   }
 }
```

The method `compileFilter`, which is not presented here, transforms a JavaScript boolean expression into SPARQL reusing a similar normalization as PLinq.

The `execute` method is defined as:

```
execute: function (sparql) {
 return DBpediaEvaluate ( sparqlToString (sparql) )
}
```

It generates the concrete SPARQL query, *i.e.* a string representing a well formed SPARQL expression.

For instance, the SPARQL code generated by the compilation of the array comprehension presented Section 2 is as follows:

```
SELECT ?city_name {
 ?p a <http://xmlns.com/foaf/0.1/Person> .
 ?p <http://xmlns.com/foaf/0.1/name> ?p_name .
 ?p dbpedia-owl:birthPlace ?p_birthPlace .
 ?city a dbpedia-owl:Place .
 ?city dbpprop:name ?city_name .
 FILTER ( ?p_name == "Barack Obama"@en
       && ?p_birthPlace == ?city )
}
```

After generating the SPARQL query, the `execute` method sends it to the DBpedia endpoint for evaluation (the `DBpediaEvaluate` function).

4. Concluding Remarks

Comprehensions help programmers writing code involving data queries as they relieve them from specifying details about the implementation of the data and its source. It is a step towards solving the *impedance mismatch problem*. Mainly inspired by the recipe proposed by Links [6] and PLinq [4], we used comprehension to transform Web sites as a data source, to make them peers of relational databases. In this short paper, we have shown how to use JavaScript array comprehensions over several data sources. We have presented a generic plugin architecture that permits users to create plugins over any data sources. Additional details about the implementation will be given in a forthcoming paper.

Currently only simplistic compilation techniques are used to compile queries. In particular, they are unable to efficiently handle querying over multiple data sources. This is left for future work.

References

[1] G. Copeland and D. Maier. Making smalltalk a database system. In *ACM Sigmod Record*, vol. 14 no 2, pages 316-325, 1984.

[2] E. Meijer, B. Beckman and G. Bierman. LINQ: Reconciling Object, Relations and XML in the .NET Framework. In *SIGMOD '06*, page 706, 2006.

[3] E. Meijer. There is no impedance mismatch:(language integrated query in Visual Basic 9). In *OOPSLA '06*, pages 710-711, 2006

[4] J. Cheney, S. Lindley and P. Wadler. A practical theory of language-integrated query. In *Proceedings of the 18th ACM SIGPLAN international conference on Functional programming*, pages 403-416, 2013

[5] D. Syme, K. Battocchi, K. Takeda and al. Strongly-Typed Language Support for Internet- Scale Information Sources. In *Technical Report MSR-TR-2012-101, Microsoft Research*, 2012

[6] E. Cooper, S. Lindley, P. Wadler and J. Yallop. Links: Web programming without tiers. In *Formal Methods for Components and Objects*, pages 266-296, 2007

[7] M. Serrano, E. Gallesio and F. Loitsch. Hop: a language for programming the web 2.0. In *OOPSLA Companion*, pages 975-985, 2006

[8] P. Buneman, L. Libkin, D. Suciu, V. Tannen and L. Wong. Comprehension syntax. In *ACM Sigmod Record*, vol. 23, no 1, pages 87-96, 1994

[9] P. Trinder. Comprehensions, a Query Notation for DBPLs. In *DBPL*, pages 55-68, 1991

[10] P. Trinder and P. Wadler. List comprehensions and the relational calculus. 1999

[11] E. F. Codd. A relational model of data for large shared data banks. In *Communications of the ACM*, vol. 26, no 1, pages 64-69, 1983

[12] Hibernate ORM. http://hibernate.org/orm/ Accessed: 2015-03-10

[13] L. Wong. Normal Forms and Conservative Properties for Query Languages over Collection Types. In *PODS '93*, pages 26-36, 1993

[14] S. Boag, D. Chamberlin, M. F. Fernndez, D. Florescu, J. Robie, J. Simon and M. Stefanescu. XQuery 1.0: An XML query language. 2002

[15] D. Kitchin, A. Quark, W. Cook and J. Misra. The Orc programming language. In *Formal techniques for Distributed Systems*, pages 1-25, 2009

[16] E. PrudHommeaux, A. Seaborne and others. SPARQL query language for RDF. In. *W3C recommendation*, vol. 15, 2008

[17] D. D. Chamberlin and R. F. Boyce. SEQUEL: A structured English query language. In *Proceedings of the 1974 ACM SIGFIDET (now SIGMOD) workshop on Data description, access and control*, pages 249-264, 1974