Regular Path Expression for Querying Semistructured Data - Implementation in Prolog

Markus Schatten, Krešimir Ivković
University of Zagreb
Faculty of Organization and Informatics
Pavlinska 2, 42000 Varaždin, Croatia
{markus.schatten, kresimir.ivkovic}@foi.hr

Abstract. *We present regular path expressions (RPE) a language for querying data graphs and its context free grammar implementation in Prolog. A proof of concept parser and query tool is implemented and various usage examples are analyzed for semistructured data formats like XML and JSON.*

Keywords. regular path expressions; semistructured data; Prolog; XML; JSON

1 Introduction

With the development of the World Wide Web and especially e-Business, there was need to exchange more and more data through the network. Since these data sources are often heterogeneous (conceptually and logically) as well as complex and incomplete, the semistructured data model was introduced [1]. The advantages of the semistructured data model include its ability to represent data which cannot be easily constrained by a schema (it is often called schema-less or self-describing data model), its flexibility in terms of data transfer, its ability to easily represent structured data as well and its ability to change its structure during time.

On the other hand its greatest deficiency arises through its flexibility in terms of structure, which makes it hard to implement efficient querying and search algorithms. A number of querying languages including xQuery/xPath [3], map/reduce [5] and a number of NoSQL techniques [12, 9, 4] as well as RPE [7] were introduced to allow for flexible querying of semistructured data. Herein we will concentrate on RPE since they provide the foundation for most of the other approaches and are applicable to any semistructured type of data.

2 Semistructured Data Model

In the following we will use graph theory to present the semistructured data model.

Definition 1 *(Directed graph)*
Let V be a set of nodes (verticles) and let $B \subseteq V \times V$ be a set of directed edges (set of ordered pairs of nodes). A directed graph or digraph G is the ordered pair (V, B). For each edge $b = (v_i, v_j)$ we say that v_i is the source ($source(b)$), and v_j the destination of the edge ($dest(b)$).

Definition 2 *(Path)*
Let $G = (V, B)$ be a digraph. Every sequence of edges $b_1/b_2/\ldots/b_k$ for which it holds that $dest(b_i) = source(b_{i+1}), i = 1, 2, \ldots, k-1$ we call a path in graph G from source $source(b_1)$ to destination $dest(b_k)$. The number of edges in the path k, is called distance.

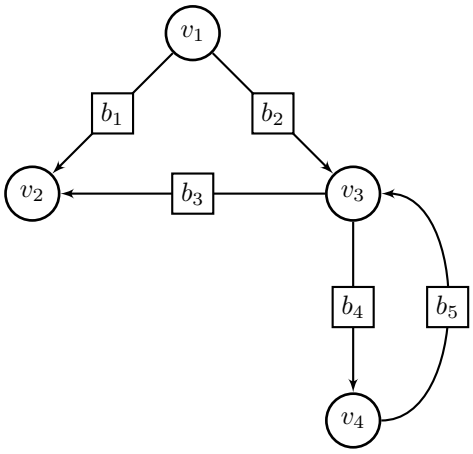
Definition 3 *(Root)*
A vertex v_k is called the root of a digraph $G = (V, B)$, if there is a path from v_k to every vertex $v_i \in V, i \neq k$.

Definition 4 *(Cycle)*
A cycle in a digraph is every path from some vertex back to itself. A graph without cycles is called acyclic.

Example 1 *Consider the digraph $G = (V, B)$, $V = \{v_1, v_2, v_3, v_4\}$, $B = \{b_1, b_2, b_3, b_4, b_5\}$:*

edge	*source*	*destination*
b_1	v_1	v_2
b_2	v_1	v_3
b_3	v_3	v_2
b_4	v_3	v_4
b_5	v_4	v_3

Graph G can be represented graphically as follows:



A few paths in G are: b_1 *;* b_2/b_3 *;* $b_2/b_4/b_5$ *;* $b_5/b_4/b_5/b_4/b_5$ *;* $b_2/b_4/b_5/b_3$

The root of G is: v_1

A few cycles in G are: b_4/b_5 *;* b_5/b_4 *;* $b_4/b_5/b_4/b_5$

In the following we shall represent graphs graphically.

Definition 5 *(Tree)*
A digraph (V, B) *is a tree iff there exists a unique path from* v_k *to* v_i *for every* $v_i \in V, i \neq k$*. Note that every tree is necessarily acyclic and has a unique root.*

Definition 6 *(Leaf)*
A vertex $v \in V$ *is a leaf of digraph* (V, B) *if there doesn't exist any edge* $b \in B$ *for which it holds that* $source(b) = v$*.*

Digraphs can be used to represent data, and in this context we shall call them data graphs. Data graphs are in most cases acyclic but there can be exceptions.

Definition 7 *(Data graph)*
A data graph is a digraph $G_D = (V, B)$ *which edges are tagged (denoted by* $o \sim b$*,* $b \in B$*), and verticles are data objects that can be:*

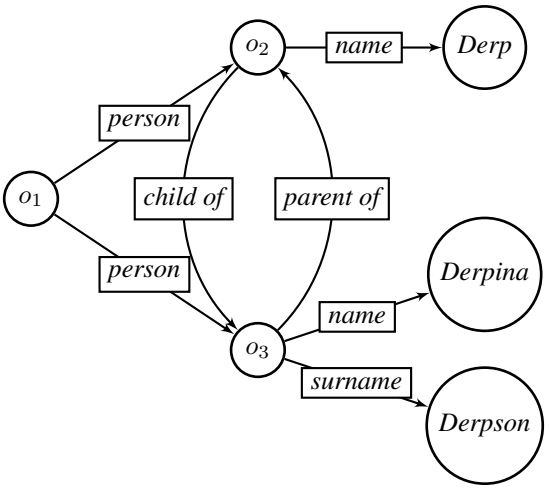
1. *atomic (leafs)*

2. *compound (have edges to other verticles)*

Every compound vertex has its object identity which is unique in the given data graph. It is sometimes useful to consider the set of verticles as an union $V = V_a \cup V_c$*, whereby* V_a *is the set of identities of atomic objects,* V_c *is the set of compound object identities, and it holds that* $V_a \cap V_c = \varnothing$*.*

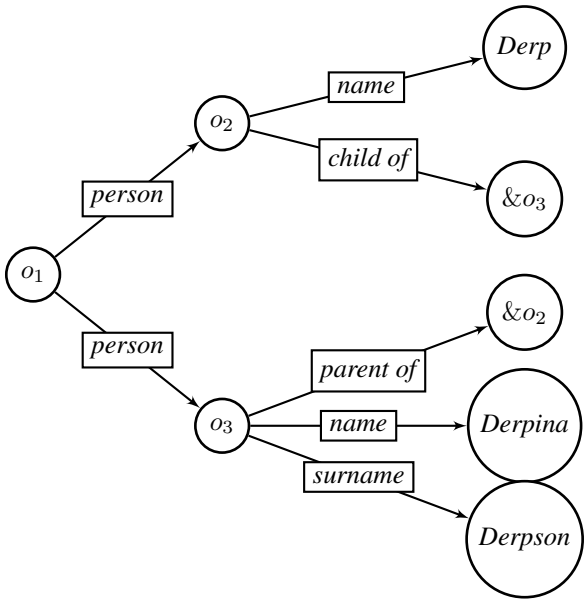
In the following we will consider data graphs that have a single root. From this view we can define data graphs as ordered triples $G_D = (V_a \cup V_c, B, r)$, whereby $r \in V_a \cup V_c$ is the root of G_D.

Example 2 *Consider the following data graph* G_1*:*



We denoted the object identities in graph G_1 *with* o_1, o_2, o_3*. Verticles Derp, Derpina i Derpson are atomic, thus* $V_a = \{Derp, Derpina, Derpson\}$*. Verticles* o_1*,* o_2 *i* o_3 *are compound, thus* $V_c = \{o_1, o_2, o_3\}$*. Edges name and child of of vertex* o_2 *denote that this vertex has adequate relations with these tags to other data objects. Note that graph* G_1 *is not a tree.*

We could have represented the graph as follows:



In this graph we used the object identities as references (prefix &), what allowed us to create a pseudo-tree of G_1*.*

3 Regular Path Expressions

In the following we shall introduce regular path expressions.

Definition 8 *(Alphabet)*
Alphabet Σ *is a finite set of symbols.*

Definition 9 *(Word)*
A word from alphabet Σ is a finite array of 0 or more symbols from Σ.

Definition 10 *(Empty word)*
A word with 0 symbols is denoted with ε and called the empty word.

Definition 11 *(Set of words over alphabet)*
Let Σ be an alphabet. With Σ^n, where $n \geqslant 0$ we denote the set of all words over alphabet Σ which have the length n. Thus the set of all words over Σ is defined as:

$$\Sigma^* = \bigcup_{n \geqslant 0} \Sigma^n$$

Similarly the set of all non-empty words over Σ is defined as:

$$\Sigma^+ = \bigcup_{n \geqslant 1} \Sigma^n$$

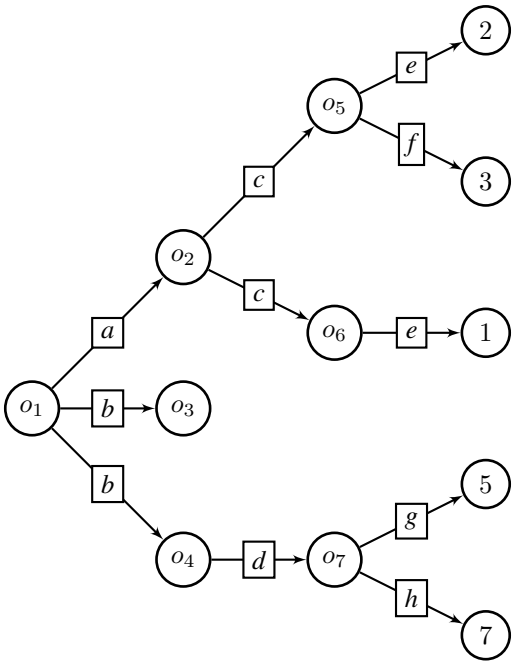
Definition 12 *(Syntax of regular path expressions)*
Let $\mathbb{S} = \{\varepsilon, \_, +, *, ?, |, -\}$ be a set of special symbols, let alphabet $\Sigma = \Theta \cup \mathbb{S}$, and let Σ^* be the set of all words defined over Σ. We define regular path expressions as the smallest set $\mathbb{RPE} \subseteq \Sigma^*$ for which it holds that:

1. $\forall s \in \Theta : s \in \mathbb{RPE}$*;*
2. $\varepsilon \in \mathbb{RPE}$ *(empty word);*
3. $\_ \in \mathbb{RPE}$ *(any character);*
4. $\forall i_1, i_2 \in \mathbb{RPE} : i_1|i_2 \in \mathbb{RPE}$ *(alternation);*
5. $\forall i_1, i_2 \in \mathbb{RPE} : i_1 i_2 \in \mathbb{RPE}$ *(concatenation);*
6. $\forall i \in \mathbb{RPE} : i? \in \mathbb{RPE}$ *(optionality);*
7. $\forall i \in \mathbb{RPE} : i+ \in \mathbb{RPE}$ *(Kleene plus, one or more repetitions);*
8. $\forall i \in \mathbb{RPE} : i* \in \mathbb{RPE}$ *(Kleene star, zero or more repetitions);*
9. $\forall s \in \Theta : s- \in \mathbb{RPE}$ *(negation).*

Specially for data graphs, the alphabet is defined as $\Theta = \{a, b, c, a_1, a_2, ...\}$ which is the set of all tags over a given data graph.[1] Usually the the symbol / (which could be interpreted as a node) is used for concatenation to easier comprehend the expression. Every path is interpreted from the root of the data graph (the first / symbol represents the root node).

We will use examples to show the semantics of $\mathbb{RPE}$.

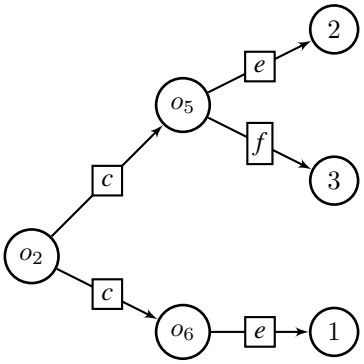
Example 3 *Consider the following data graph $G = (V, B, o_1)$:*



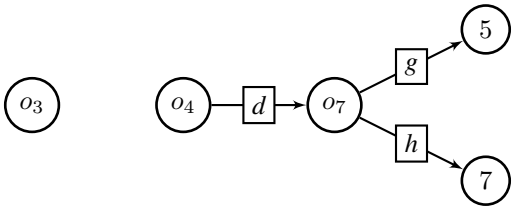
From the graph we read that the alphabet is $\Theta = \{a, b, c, d, e, f, g, h\}$. Consider the following queries over G:

$$\begin{aligned} Q_1 &: \quad /a \\ Q_2 &: \quad /b \\ Q_3 &: \quad \varepsilon \\ Q_4 &: \quad /w \end{aligned}$$

The result of query Q_1 is the following graph:



Note that the query has tagged all nodes which satisfy the path, which in this case is only node o_2. On the other hand the resulting graphs of query Q_2 are:



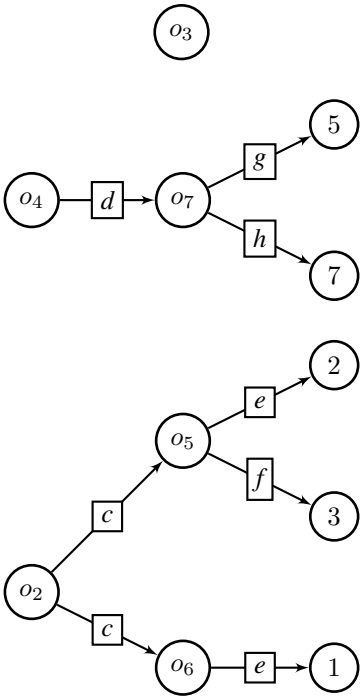
[1] Usually a schema graph is defined which constraints the possible tags that can be used, but this was left out here for sake of simplicity.

Thus in this case the nodes o_3 and o_4 did satisfy the path $/b$. The result of query Q_3 is the empty graph (denoted with $G^{\varnothing}$). This means that there is no node which can satisfy the given path. The query Q_4 is not applicable to graph G since $w \notin \Theta$.

Example 4 *Consider the following query on graph G from example 3:*

$$Q_5 : /\_$$

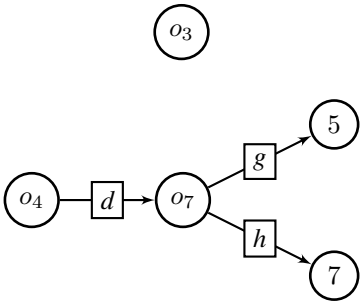
The symbol _ denotes an arbitrary sign. Thus the resulting graphs of query Q_5 are:

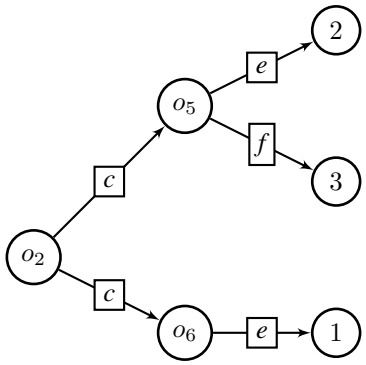


Example 5 *Let graph G as in example 3. We introduce alternation and concatenation. Consider the following queries:*

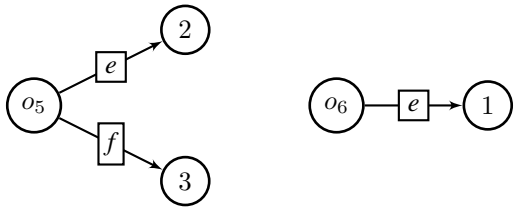
$$Q_6 : \quad /a \mid /b$$
$$Q_7 : \quad /a/c$$

The $\mid$ symbol denotes "or" and thus the result of the query is the union of results of the queries $/a$ and $/b$:

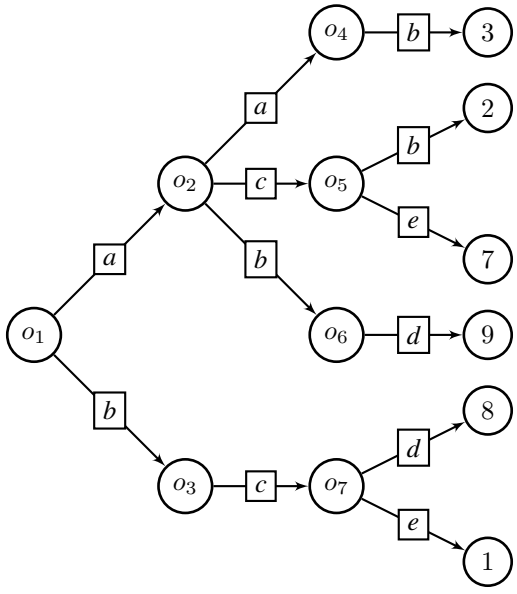




By concatenating queries we create a composition whereby from left to right the adjacent queries are applied on the results of the previous query. Thus the result of query Q_7 are the graphs:



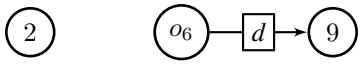
Example 6 *Let graph G_1 be defined as follows:*



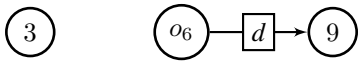
Consider the following queries:

$$Q_1 : \quad /a/c?/b$$
$$Q_2 : \quad /a + /b$$
$$Q_3 : \quad /a * /b$$
$$Q_4 : \quad /a - /c$$

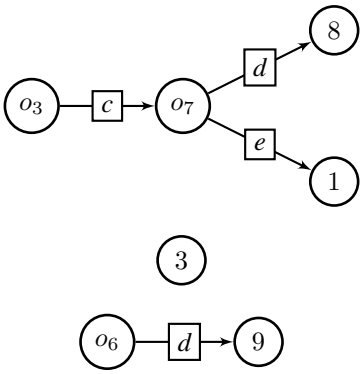
The ? operator denotes optionality (the occurrence of the previous query is optional). Thus the resulting graphs of query Q_1 are:



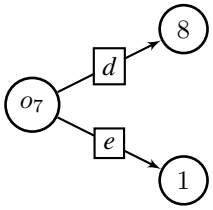
The + operator denotes one or more occurrences of the previous query. Thus the resulting graphs of query Q_2 are:



The ∗ operator denotes repetition of the previous query zero or more times. Thus the resulting graphs of query Q_3 are.



The − operator denotes negation, meaning that the previous query shouldn't return any results. Thus the result of query Q_4 is:



4 Prolog Implementation

We implemented ℝℙ𝔼 using XSB Prolog [11] with a little help from Python to implement a user interface. The syntax of ℝℙ𝔼 was implemented using a context free grammar which is shown bellow:

Listing 1: Context free grammar of ℝℙ𝔼

```
operators(
  [ '?', '+',
    '*', '!',
    '_', '/',
    '(', ')',
    '|' ] ).
symbol( X ) :-
  operators( O ),
  not( member( X, O ) ),
  atomic( X ).
simple( X ) :-
  functor( X, F, _ ),
  member( F,
    [ sym, any, empty, exp ] ).
exp( sym( S ) ) --> sym( S ).
sym( S ) --> [ S ], { symbol( S ) }.
exp( empty( '' ) ) --> [ '' ].
exp( any( '_' ) ) --> [ '_' ].
exp( alter( X, Y ) ) -->
  exp( X ), [ '|' ],
  exp( Y ).
exp( conc( X, Y ) ) -->
  exp( X ), [ '/' ],
  exp( Y ),
  {
    not( X = root( _ ) ),
    not( Y = conc( _, _ ) )
  }.
exp( opt( X ) ) -->
  exp( X ), [ '?' ],
  {
    simple( X )
  }.
exp( star( X ) ) -->
  exp( X ), [ '*' ],
  {
    simple( X )
  }.
exp( plus( X ) ) -->
  exp( X ), [ '+' ],
  {
    simple( X )
  }.
exp( neg( X ) ) -->
  exp( X ), [ '!' ],
  {
    simple( X )
  }.
exp( exp( X ) ) -->
  [ '(' ], exp( X ), [ ')' ].
exp( root( X ) ) -->
  [ '/' ], exp( X ).
```

This implementation of the grammar allows us to construct a syntax tree of ℝℙ𝔼 automatically providing only a list of symbols. As an example consider the following Prolog query:

Listing 2: Abstract syntax tree construction

```
|?- exp(T,['/','a','+','/','b'],[]).

T = root(conc(plus(sym(a)),sym(b)))
```

This structure is the abstract syntax tree of expression $/a + /b$.

The semantics of ℝℙ𝔼 were implemented by using a set of rules. The following listing shows an example of one such rule.

Listing 3: Example of concatenation matching rule

```
match(conc(X,Y),Doc,Res,RootDoc) :-
  match(X,Doc,Res,RootDoc),
```

```
    match(Y,Doc,empty([]),RootDoc).
match(conc(X,Y),Doc,Res,RootDoc) :-
    match(X,Doc,Res1,RootDoc),
    compound(Res1),
    arg(1,Res1,List),
    member(Child,List),
    match(Y,Child,Res,RootDoc),
    Res \= empty([]).
```

In order to test our application we implemented additional parsers for XML and JSON formats, which translate them into parsable Prolog structures.

5 Usage Examples

The implemented program is a simple command line tool which takes a XML or JSON document and a $\mathbb{RPE}$ as its parameters and returns the result of the query. Consider the following XML document:

Listing 4: Example of XML document (`test.xml`)

```
<?xml version="1.0"?>
<ro>
<a>
  <a>
    <b>3</b>
  </a>
  <c>
    <b>2</b>
    <e>7</e>
  </c>
  <b>
    <d>9</d>
  </b>
</a>
<b>
  <c>
    <d>8</d>
    <e>1</e>
  </c>
</b>
</ro>
```

An equivalent JSON document would look as follows:

Listing 5: Example of JSON document (`test.js`)

```
{
  "ro": [{
    "a": [{
      "a": [{ "b":3 }]
    },
    {
      "c": [{ "b":2, "e":7 }]
    },
    {
      "b": [{ "d":9 }]
    }]
  },
  {
    "b": [{
      "c": [{ "d":8, "e":1 }]
    }]
  }]
}
```

To query the XML document we type:

Listing 6: Example query $/ro/a - /c$ (XML format)

```
$ ./rpe test.xml '/ro/a!/c'

<?xml version="1.0" ?>
<c>
        <d>
                8
        </d>
        <e>
                1
        </e>
</c>
```

Which is the expected output. Equivalently if we issue the same query against the JSON document, we get:

Listing 7: Example query $/ro/a - /c$ (JSON format)

```
$ ./rpe test.js '/ro/a!/c'

{"c": {"e": "1", "d": "8"}}
```

6 Related work

There have been a number of implementations which deal with querying semistructured data using some form of regular expressions or regular path expressions. For example Fxgrep is an XML querying tool similar to XPath [2]. StruQL is a querying language for XML which has some features similar to SQL but allows the usage of regular expressions in paths [10]. The GLEEN path expression library allows the usage or regular expressions in SPARQL [6]. GraphGrep is a graph querying tool which uses their internal dataset file format [8]. None of the above however allows for querying both XML and JSON format.

7 Conclusion

The implementation of $\mathbb{RPE}$ in Prolog by using context free grammars is straight-forward as we have shown in this paper. By using a graph theoretic approach to formalizing data graphs and $\mathbb{RPE}$ we were able to implement a simple querying tool for XML and JSON formats. Our next steps include implementation of other formats (for example ZODB & SPARQL), the extension of $\mathbb{RPE}$ with formulas as well as formalizing fuzzy $\mathbb{RPE}$.

References

[1] ABITEBOUL, S., BUNEMAN, P., AND SUCIU, D. *Data on the Web - From Relations to Semistructured Data and XML*. Morgan Kaufmann Publishers, 2000.

[2] BERLEA, A. Fxgrep - a functional xml querying tool. 2005.

[3] BOAG, S., CHAMBERLIN, D., FERNÁNDEZ, M. F., FLORESCU, D., ROBIE, J., AND SIMÉON, J. Xquery 1.0: An xml query language (second edition), December 2010.

[4] CATTELL, R. Scalable sql and nosql data stores. *SIGMOD Rec. 39*, 4 (May 2011), 12–27.

[5] DEAN, J., AND GHEMAWAT, S. Mapreduce: simplified data processing on large clusters. In *Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation - Volume 6* (Berkeley, CA, USA, 2004), OSDI'04, USENIX Association, pp. 10–10.

[6] DETWILER, L. T., SUCIU, D., AND BRINKLEY, J. F. Regular paths in SparQL: Querying the NCI thesaurus. *AMIA ... Annual Symposium proceedings / AMIA Symposium. AMIA Symposium* (2008), 161–165.

[7] FLORESCU, D., AND KOSSMANN, D. A performance evaluation of alternative mapping schemes for storing XML data in a relational database. Tech. rep., 1999.

[8] GIUGNO, R., AND SHASHA, D. Graphgrep. 2002.

[9] LEAVITT, N. Will NoSQL databases live up to their promise? *Computer 43*, 2 (Feb. 2010), 12–14.

[10] MORK, P. Struql. 2003.

[11] SAGONAS, K. F., SWIFT, T., AND WARREN, D. S. The xsb programming system. In *Workshop on Programming with Logic Databases (Informal Proceedings), ILPS* (1993), p. 164.

[12] STONEBRAKER, M. Sql databases v. nosql databases. *Communications of the ACM 53*, 4 (2010), 10.