

RDF/OWL and SPARQL instead of NoSQL databases

Philip Cannata (Oracle and the University of Texas), Isabella Bhardwaj (University of Texas), Michael Porras (Rackspace), Steven Spohrer (University of Texas and Samsung Research), Andrew Oldag, Francisco Garcia

1. Introduction

NoSQL database systems, especially MongoDB (MongoDB, 2014), have been gaining notoriety over the past several years purportedly for several reason 1) their data model (e.g., JSON) and its “schema-less” flexibility is a better match for data-driven web applications; 2) they are well suited for Database-as-a-Service (DBaaS) implementations (mongolab, 2014); and 3) they can scale across small, inexpensive systems, i.e., they can “scale out”. However, the underlying, “flexible” data model for these systems is very similar to technology that has a very long history in the form of Entity-Attribute-Value (EAV) database systems (Stead, Hammond, & Straube, 1982) (McDonald, Blevins, Tierney, & Martin, 1988).

EAV technology has evolved into the standards-based RDF/OWL and SPARQL technology of today and this paper will show that RDF/OWL and SPARQL are practical alternatives to current NoSQL databases for building data-driven web applications and for DBaaS. (Scale-out won’t be discussed in this paper because it is believed to be orthogonal to the discussion.)

As a case study to demonstrate the capabilities of RDF/OWL and SPARQL, this paper will present an implementation of a simple, flask-based (Flask, 2014) book website developed using the MongoDB NoSQL database system and the same website developed with RDF/OWL and SPARQL embedded in a system called ReL¹. The paper will also show how to build DBaaS applications using RESTful ReL. Both of these examples will show that RDF/OWL and SPARQL are very well suited for these types of applications.

¹ ReL (Relation Language) is a python-based data management system that uses RDF/OWL and SPARQL as its tuple manager. ReL also allows data manipulation and retrieval using a mix and match of many different higher-level data models, including the Relational Model, a Semantic Model based upon the work of Hammer and McLeod (Hammer & McLeod, 1981), and the OO python model. The ReL Relational Model (which is automatically translated to RDF/OWL and SPARQL) is used in this paper. However, because ReL is a python-based system, it’s trivial to also support JSON by translating it into one of the other supported data models. This has been done in several ReL applications.

These examples also demonstrate that a major weakness of NoSQL databases – that there is no universal query language - could be overcome by standardizing on the SPARQL query language.

This paper also shows that, given a suitable implementation of an RDF triple-store (e.g. (Oracle Graph, 2014)), support for standard transaction capabilities (i.e., read committed, serializable, and ACID) can be made available to NoSQL-like applications. This addresses another major weakness of NoSQL databases because they do not provide this level of transaction support. MongoDB, for example only provides ...

Figure 1 shows the main menu for the book website.

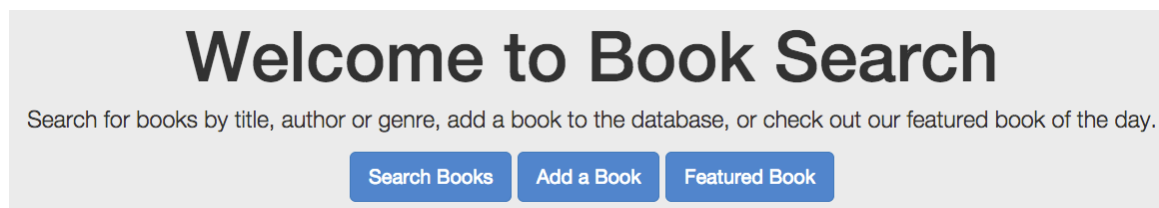


Figure 1

2. Inserting data into the book website database

Figure 2 shows the menu for adding a book in this web application.

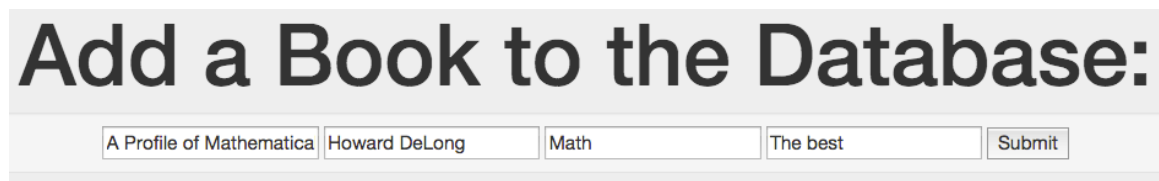


Figure 2

When the submit button is clicked, the following code is executed.

A. For the MongoDB application

First, a MongoDB connection is established:

```
from flask import Flask, render_template, redirect, request
import pymongo
```

```
app = Flask(__name__, static_url_path = "")
connection_string = "mongodb://127.0.0.1"
```

```

connection = pymongo.MongoClient(connection_string)
database = connection.books
books = database.mybooks

```

Then, the insert is done using the MongoDB insert API:

```

@app.route('/add/', methods=['GET', 'POST'])
def add():
    if request.method == 'POST':
        new_data = {k : v for k, v in request.form.items()}
        books.insert(new_data)
        return render_template('add.html', alert =
            "success")
    else:
        return render_template('add.html', alert="")

```

B. For the ReL application

First, a connection to an Oracle RDF datastore is established:

```

from flask import Flask, render_template, redirect, request

app = Flask(__name__, static_url_path = "")
conn = connectTo 'jdbc:oracle:thin:@host:1521:orcl' 'user'
        'password' 'rdf_mode' 'bookApp'

```

Then, the insert is done using a standard SQL insert; however, no table named “books” was created beforehand, so this is “scheme-less,” just like MongoDB:

```

@app.route('/add/', methods=['GET', 'POST'])
def add():
    if request.method == 'POST':
        new_data = {k : v for k, v in request.form.items()}
        values = (str(new_data['title']),
            str(new_data['author']), str(new_data['genre']),
            str(new_data['description']))
        SQL on conn """insert into books(title, author,
        genre, description) values"""values
        return render_template('add.html', alert =
            "success")
    else:
        return render_template('add.html', alert="")

```

Behind the scenes, ReL converts the SQL insert into a series of several RDF/OWL insert statements as follows²:

² URIs are abbreviated to help with readability.

```

BEGIN
commit ;
set transaction isolation level serializable ;
INSERT INTO BOOK_DATA VALUES ( BOOK_APP_SQNC.nextval,
SDO_RDF_TRIPLE_S('FALL2014_CS347_PROF:<owl>', 'owl#89',
'owl#title', '"A Profile of Mathematical
Logic"^^xsd:string'));
INSERT INTO BOOK_DATA VALUES ( BOOK_APP_SQNC.nextval,
SDO_RDF_TRIPLE_S('FALL2014_CS347_PROF:<owl>', 'owl#title',
'rdf:type', 'owl:DatatypeProperty'));
INSERT INTO BOOK_DATA VALUES ( BOOK_APP_SQNC.nextval,
SDO_RDF_TRIPLE_S('FALL2014_CS347_PROF:<owl>', 'owl#title',
'rdfs:domain', 'owl#books'));
INSERT INTO BOOK_DATA VALUES ( BOOK_APP_SQNC.nextval,
SDO_RDF_TRIPLE_S('FALL2014_CS347_PROF:<owl>', 'owl#title',
'rdf:range', 'rdfs:xsd:string'));
INSERT INTO BOOK_DATA VALUES ( BOOK_APP_SQNC.nextval,
SDO_RDF_TRIPLE_S('FALL2014_CS347_PROF:<owl>', 'owl#title',
'rdf:type', 'owl:FunctionalProperty'));
INSERT INTO BOOK_DATA VALUES ( BOOK_APP_SQNC.nextval,
SDO_RDF_TRIPLE_S('FALL2014_CS347_PROF:<owl>', 'owl#89',
'owl#author', '"Howard DeLong"^^xsd:string'));

INSERT INTO BOOK_DATA VALUES ( BOOK_APP_SQNC.nextval,
SDO_RDF_TRIPLE_S('FALL2014_CS347_PROF:<owl>', 'owl#author',
'rdf:type', 'owl:DatatypeProperty'));
INSERT INTO BOOK_DATA VALUES ( BOOK_APP_SQNC.nextval,
SDO_RDF_TRIPLE_S('FALL2014_CS347_PROF:<owl>', 'owl#author',
'rdfs:domain', 'owl#books'));
INSERT INTO BOOK_DATA VALUES ( BOOK_APP_SQNC.nextval,
SDO_RDF_TRIPLE_S('FALL2014_CS347_PROF:<owl>', 'owl#author',
'rdf:range', 'rdfs:xsd:string'));
INSERT INTO BOOK_DATA VALUES ( BOOK_APP_SQNC.nextval,
SDO_RDF_TRIPLE_S('FALL2014_CS347_PROF:<owl>', 'owl#author',
'rdf:type', 'owl:FunctionalProperty'));
INSERT INTO BOOK_DATA VALUES ( BOOK_APP_SQNC.nextval,
SDO_RDF_TRIPLE_S('FALL2014_CS347_PROF:<owl>', 'owl#89',
'owl#genre', '"Math"^^xsd:string'));

INSERT INTO BOOK_DATA VALUES ( BOOK_APP_SQNC.nextval,
SDO_RDF_TRIPLE_S('FALL2014_CS347_PROF:<owl>', 'owl#genre',
'rdf:type', 'owl:DatatypeProperty'));
INSERT INTO BOOK_DATA VALUES ( BOOK_APP_SQNC.nextval,
SDO_RDF_TRIPLE_S('FALL2014_CS347_PROF:<owl>', 'owl#genre',
'rdfs:domain', 'owl#books'));

```

```

INSERT INTO BOOK_DATA VALUES ( BOOK_APP_SQNC.nextval,
SDO_RDF_TRIPLE_S('FALL2014_CS347_PROF:<owl>', 'owl#genre',
'rdf:range', 'rdfs:xsd:string'));
INSERT INTO BOOK_DATA VALUES ( BOOK_APP_SQNC.nextval,
SDO_RDF_TRIPLE_S('FALL2014_CS347_PROF:<owl>', 'owl#genre',
'rdf:type', 'owl:FunctionalProperty'));
INSERT INTO BOOK_DATA VALUES ( BOOK_APP_SQNC.nextval,
SDO_RDF_TRIPLE_S('FALL2014_CS347_PROF:<owl>', 'owl#89',
'owl#description', '"The best"^^xsd:string'));

INSERT INTO BOOK_DATA VALUES ( BOOK_APP_SQNC.nextval,
SDO_RDF_TRIPLE_S('FALL2014_CS347_PROF:<owl>',
'owl#description', 'rdf:type', 'owl:DatatypeProperty'));
INSERT INTO BOOK_DATA VALUES ( BOOK_APP_SQNC.nextval,
SDO_RDF_TRIPLE_S('FALL2014_CS347_PROF:<owl>',
'owl#description', 'rdfs:domain', 'owl#books'));
INSERT INTO BOOK_DATA VALUES ( BOOK_APP_SQNC.nextval,
SDO_RDF_TRIPLE_S('FALL2014_CS347_PROF:<owl>',
'owl#description', 'rdf:range', 'rdfs:xsd:string'));
INSERT INTO BOOK_DATA VALUES ( BOOK_APP_SQNC.nextval,
SDO_RDF_TRIPLE_S('FALL2014_CS347_PROF:<owl>',
'owl#description', 'rdf:type', 'owl:FunctionalProperty'));
END ;
/

```

Notice that some of these insert statements insert appropriate OWL schema information into the RDF tuple-store, e.g., for the “title” attribute, the following is inserted:

```

INSERT INTO BOOK_DATA VALUES ( BOOK_APP_SQNC.nextval,
SDO_RDF_TRIPLE_S('FALL2014_CS347_PROF:<owl>', 'owl#title',
'rdf:type', 'owl:DatatypeProperty'))
INSERT INTO BOOK_DATA VALUES ( BOOK_APP_SQNC.nextval,
SDO_RDF_TRIPLE_S('FALL2014_CS347_PROF:<owl>', 'owl#title',
'rdfs:domain', 'owl#books'))
INSERT INTO BOOK_DATA VALUES ( BOOK_APP_SQNC.nextval,
SDO_RDF_TRIPLE_S('FALL2014_CS347_PROF:<owl>', 'owl#title',
'rdf:range', 'rdfs:xsd:string'))
INSERT INTO BOOK_DATA VALUES ( BOOK_APP_SQNC.nextval,
SDO_RDF_TRIPLE_S('FALL2014_CS347_PROF:<owl>', 'owl#title',
'rdf:type', 'owl:FunctionalProperty'))

```

This schema information is used when querying the book database.

3. Searching the book website database

Figure 3 shows the menu for searching for a book in this web application.

Search The Database:

Figure 3

When the submit button is clicked, the following code is executed:

A. For the MongoDB application

The search is done using the MongoDB “find” API:

```
@app.route('/search/', methods=['GET', 'POST'])
def search():
    if request.method == 'POST':
        query = request.form['query']
        return render_template('search.html', posting=True,
                               query=query, title_results = books.find({
                                   'title':query}),author_results = books.find({
                                   'author':query}))
    else:
        return render_template('search.html', posting =
                               False)
```

B. For the ReL application

The ReL search is done using standard SQL select statements as follows (the SQL select statements return a python tuple of tuples):

```
@app.route('/search/', methods=['GET', 'POST'])
def search():
    #Return results for titles, authors and genres that
    match the search query
    if request.method == 'POST':
        query = request.form['query']

        titles = SQL on conn """select title, author from
            books where title = """query"""
        authors = SQL on conn """select title, author from
            books where author = """query"""
        title_dict = {}
        num = 0
        for j in titles :
```

```

        title_dict.update({'Key' + str(num) : {'title'
        : j[0], 'author' : j[1]}})
        num += 1
    author_dict = {}
    num = 0
    for j in authors :
        author_dict.update({'Key' + str(num) : {'title'
        : j[0], 'author' : j[1]}})
        num += 1

    genre_dict = {}

    no_results = title_dict == 0 and author_dict == 0
        and genre_dict == 0
    return render_template('search.html', posting=True,
        query=query, no_results=no_results,
        title_results=title_dict,
        author_results=author_dict,
        genre_results=genre_dict)

else:
    return render_template('search.html', posting =
        False)

```

Behind the scenes, ReL converts the SQL select statements into SPARQL statements, as shown below:

```

SELECT v1 "title", v2 "author"
FROM TABLE(SEM_MATCH('SELECT * WHERE {
    ?s1 rdf:type :books .
    OPTIONAL { ?s1 :title ?v1 }
    OPTIONAL { ?s1 :author ?v2 }
    ?s1 :title ?f1 .
    FILTER(?f1 = "Howard DeLong") }' ,
SEM_MODELS('FALL2014_CS347_PROF'), null,
SEM_ALIASES( SEM_ALIAS('','http://www.example.org/people.owl#')), null) )

SELECT v1 "title", v2 "author"
FROM TABLE(SEM_MATCH('SELECT * WHERE {
    ?s1 rdf:type :books .
    OPTIONAL { ?s1 :title ?v1 }
    OPTIONAL { ?s1 :author ?v2 }
    ?s1 :author ?f1 .
    FILTER(?f1 = "Howard DeLong") }' ,
SEM_MODELS('FALL2014_CS347_PROF'), null,

```

```
SEM_ALIASES( SEM_ALIAS('',
'http://www.example.org/people.owl#')), null) )
```

4. Transaction support in ReL

The SPARQL statements shown above can be included in a regular Oracle transaction (see the example below). The same is true for all other database operations in ReL. So, ReL can provide traditional transaction support (i.e., read committed, serializable, and ACID) for all of its database operations. (Notice in Section 2 above, the inserts were wrapped in PL/SQL BEGIN and END statements, which makes the inserts ACID). One of the major criticisms of NoSQL databases is that they don't provide traditional transaction support. ReL does not suffer from this problem.

```
commit ;
set transaction isolation level serializable ;

SELECT v1 "title", v2 "author"
  FROM TABLE(SEM_MATCH('SELECT * WHERE {
    ?s1 rdf:type :books .
    OPTIONAL { ?s1 :title ?v1 }
    OPTIONAL { ?s1 :author ?v2 }
    ?s1 :title ?f1 .
    FILTER(?f1 = "Howard DeLong") }' ,
SEM_MODELS('FALL2014_CS347_PROF'), null,
SEM_ALIASES( SEM_ALIAS('',
'http://www.example.org/people.owl#')), null) );

SELECT v1 "title", v2 "author"
  FROM TABLE(SEM_MATCH('SELECT * WHERE {
    ?s1 rdf:type :books .
    OPTIONAL { ?s1 :title ?v1 }
    OPTIONAL { ?s1 :author ?v2 }
    ?s1 :author ?f1 .
    FILTER(?f1 = "Howard DeLong") }' ,
SEM_MODELS('FALL2014_CS347_PROF'), null,
SEM_ALIASES( SEM_ALIAS('',
'http://www.example.org/people.owl#')), null) );
```

5. DBaaS - RESTful ReL

ReL can also be run as a RESTful (Fielding, 2000) server, which means ReL SQL calls (e.g., `""select title, author from books where title = ""query""""""""`, which was discussed above) can be embedded in any environment that supports the CURL function. For instance, we use RESTful ReL in R to access data and convert it to R data frames for analysis. The same “data model to RDF/OWL and SPARQL”

translations that were discussed above can be used with Restful ReL. Here's how ReL can be invoked from R to query the standard Oracle emp table:

```
d = getURL( URLencode('host:5000/rest/native/?query =
"select * from emp"), httpheader =
c(DB='jdbc:oracle:thin:@host:1521:orcl', USER='user',
PASS='password', MODE='rdf_mode', MODEL='Fall2014',
returnFor = 'R'), verbose = TRUE)
```

The "returnFor = 'R'" httpheader parameter value above directs ReL to return data in the following format:

```
d
"list(c('COMM', 'HIREDATE', 'JOB', 'DEPTNO', 'SAL',
'ENAME', 'MGR', 'EMPNO'), list(c('NULL', 1400, 'NULL',
'NULL', 500, 'NULL', 300, 'NULL', 'NULL', 'NULL', 'NULL',
'NULL', 'NULL', 'NULL'),c('23-JAN-1982', '28-SEP-1981', '1-
MAY-1981', '3-DEC-1981', '22-FEB-1981', '9-JUN-1981', '20-
FEB-1981', '8-SEP-1981', '12-JAN-1983', '09-DEC-1982', '17-
NOV-1981', '17-DEC-1980', '3-DEC-1981', '2-APR-
1981'),c('CLERK', 'SALESMAN', 'MANAGER', 'ANALYST',
'SALESMAN', 'MANAGER', 'SALESMAN', 'SALESMAN', 'CLERK',
'ANALYST', 'PRESIDENT', 'CLERK', 'CLERK', 'MANAGER'),c(10,
30, 30, 20, 30, 10, 30, 30, 20, 20, 10, 20, 30, 20),c(1300,
1250, 2850, 3000, 1250, 2450, 1600, 1500, 1100, 3000, 5000,
800, 950, 2975),c('MILLER', 'MARTIN', 'BLAKE', 'FORD',
'WARD', 'CLARK', 'ALLEN', 'TURNER', 'ADAMS', 'SCOTT',
'KING', 'SMITH', 'JAMES', 'JONES'),c(7782, 7698, 7839,
7566, 7698, 7839, 7698, 7698, 7788, 7566, 'NULL', 7902,
7698, 7839),c(7934, 7654, 7698, 7902, 7521, 7782, 7499,
7844, 7876, 7788, 7839, 7369, 7900, 7566)))"
```

Then, the data can be converted to an R data frame using the following two commands:

```
df <- data.frame(eval(parse(text=substring(d,1))) [2])
colnames(df) <- unlist(eval(parse(text=substring(d,1))) [1])
```

Finally, head(df) results in the following:

```
head(df)
  COMM  HIREDATE      JOB DEPTNO  SAL  ENAME  MGR EMPNO
1 NULL 23-JAN-1982   CLERK     10 1300 MILLER 7782  7934
2 1400 28-SEP-1981 SALESMAN     30 1250 MARTIN 7698  7654
```

3	NULL	1-MAY-1981	MANAGER	30	2850	BLAKE	7839	7698
4	NULL	3-DEC-1981	ANALYST	20	3000	FORD	7566	7902
5	500	22-FEB-1981	SALESMAN	30	1250	WARD	7698	7521
6	NULL	9-JUN-1981	MANAGER	10	2450	CLARK	7839	7782

6. Summary

In an InfoWorld article (Oliver, 2014), the author claims, “the time for NoSQL standards is now”. But, RDF/OWL and SPARQL are standards that exist now and this paper has demonstrated that they are perfectly well suited for building NoSQL type data-driven web applications and DBaaS applications. What’s more, unlike other NoSQL systems, ReL supports the standard, read committed, and serializable transaction processing capabilities of traditional database management systems. So, maybe “the time for NoSQL to use existing standards is now”.

The Oracle implementation of RDF/OWL and SPARQL (Oracle Graph, 2014) was used for the applications in this paper; however, any proper implementation of these standards could be used instead.

This paper did not discuss the “scale out rather than scale up” proposition of NoSQL databases, but that debate has a decades long history and needs no more discussion here. However, there is no reason that RDF/OWL and SPARQL could not be used just as effectively in a “scale out” system as opposed to Oracle’s “scale up” system.

7. Bibliography

Fielding, R. (2000). *REST*. Retrieved from http://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm

Flask. (2014). Retrieved from Flask web development, one drop at a time: <http://flask.pocoo.org/>

Hammer, M., & McLeod, D. (1981). Database Description with SDM: A Semantic Database Model. *ACM Transaction of Database*, 6 (3).

McDonald, C., Blevins, L., Tierney, W., & Martin, D. (1988). The Regenstrief Medical Records. *MD Computing* (5(5)), 34-47.

MongoDB. (2014). Retrieved from MongoDB: <http://www.mongodb.org>

mongolab. (2014). Retrieved from <https://mongolab.com/>

Oliver, A. (2014). *The time for NoSQL standards is now*. Retrieved from InfoWorld: <http://www.infoworld.com/article/2615807/nosql/the-time-for-nosql-standards-is-now.html>

Oracle Graph. (2014). Retrieved from
<http://www.oracle.com/us/products/database/options/spatial/overview/index.html>

Stead, W., Hammond, W., & Straube, M. (1982, November). A Chartless Record—Is It Adequate? *Proceedings of the Annual Symposium on Computer Application in Medical Care*, 89-94.