RDF/OWL and SPARQL instead of NoSQL databases

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

Abstract. NoSQL database technology has become increasingly popular in the data management field over the past few years with more than 25 percent adoption claimed in 2014[1]. This phenomenal growth has occurred in spite of some striking shortcomings in NoSQL technologies. In particular, NoSQL databases:

1. currently lack a standard query language
2. do not support "joins", so they encourage applications to denormalize their data[2] and in the case of Cassandra, to also create and manage Materialized Views (Apache Cassandra, 2014), with all of the headaches that entails
3. are reinventing the wheel with proprietary features like MongoDB aggregation (MongoDB Aggregation, 2014)
4. do not support ACID (Transactions, 2011) transaction processing[3] as found in modern data management systems because of their perceived need for a special kind of "scaleout"[4] (i.e., so called "inexpensive" scaleout).

[1] 2014 Forrester Research, Inc.

[2] MongoDB allows for joins to be done in the application by the use of "Manual References" or DBRefs (MongoDBRefs) but this is functionality that should be done in the database system not in the application.

[3] The transaction issues found in NoSQL database systems were first addressed in the early 1990s in work on "Relaxed Transaction Processing" in the Carnot Project at MCC (Cannata, 1991) (Carnot Home Page) (Paul Attie, 1993) (Munindar Singh, 1994) and has recently led to a decade long debate of ACID vs. BASE vs. SALT transaction processing (Brewer, 2000), (Seth Gilbert, 2002), (Chao Xie, 2014). See Section 3 "Transaction Support" of this paper for more details.

[4] "Scalability is the ability of an application to efficiently use more resources in order to do more useful work. For example, an application that can service four users on a single-processor system may be able to service 15 users on a four-processor system. In this case, the application is scalable. If adding more processors doesn't increase the number of users serviced (if the application is single threaded, for example), the application isn't scalable.

There are two kinds of scalability: scaleup and **scaleout**. Scaleup means scaling to a bigger, more powerful server—going from a four-processor server to a 128-processor, for example. This is the most common way for databases to scale. When your database runs out of resources on your current hardware, you go out and buy a bigger box with more processors and more memory. Scaleup has the advantage of not requiring significant changes to the database. In general, you just install your database on a bigger box and keep running the way you always have, with more database power to handle a heavier load. **Scaleout** means expanding to multiple servers rather than a single, bigger server. Scaleout usually has some initial hardware cost advantages—eight four-processor servers generally cost less than one 32-processor server—but this advantage is often cancelled out when licensing and maintenance costs are included. In some cases, the redundancy offered by a scaleout solution is also useful from an availability perspective." (SQL Server Scale Out White Paper - Microsoft)

This paper proposes that RDF/OWL and SPARQL database technology[5] is an appropriate alternative to current NoSQL technology because:

1. SPARQL is an international standard and has all of the features required to be a standard query language for NoSQL-type databases
2. RDF/OWL and SPARQL systems support "joins" in the database, no application processing is required for joins (see Section 5 of this paper for an example)
3. SPARQL has full support for features like aggregation (see Section 5 of this paper for an example)
4. existing RDF/Owl implementations already provide ACID transaction processing support (Oracle Graph, 2014) in scaleup and scaleout configurations (Exadata, 2014).

In other words, RDF/OWL and SPARQL technology does not have the striking shortcomings of NoSQL technology.

In addition, this paper also argues that the highly touted features of NoSQL databases are easily achieved using RDF/OWL and SPARQL database technology. These features are:

1. flexible, "schema-less" data models "in which the semantics of the data are embedded within a flexible connection topology and a corresponding storage model. This provides greater flexibility for managing large data sets while simultaneously reducing the dependence on the more formal database structure imposed by the relational database " (Loshin)
2. Agile database application development, which "includes a set of software development methods focused on an iterative approach to building software (as opposed to software development methods that focus on rigorous planning and scheduling in advance). Agile development means that tasks are broken into small pieces, and evaluated and built on as they are completed, with users and other stakeholders able to frequently see small, completed results. . . . SQL databases, which require a schema defined upfront and subsequent (and costly) database migrations as schemas change, are more difficult to use with agile methods and impossible to use in a continuously integrated environment without significant additional engineering (AgileMongoDB, 2014)
3. a good fit with RESTful (Tilkov) services in web-based applications and in Database-as-a-Service (DBaaS) (Mongolab, 2014) (CloudCredo) environments.

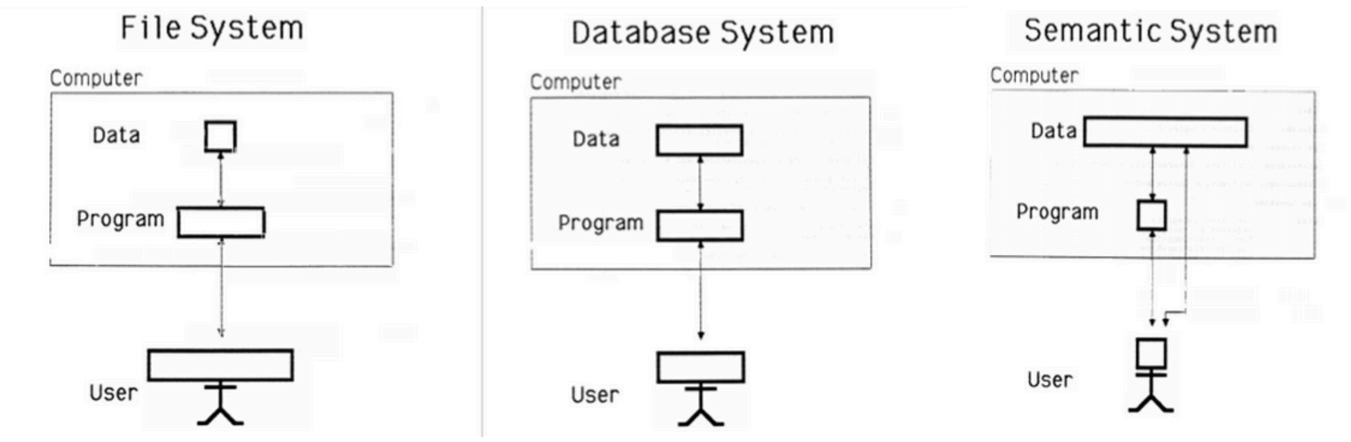
Lastly, this paper outlines the game-changing potential of Inferencing in data management systems. Inferencing is natively supported in OWL-based systems, but not available in Relational or current NoSQL databases. The capabilities of Inferencing can, at a minimum, act as a guide to a better future for Data Management where the Database Management Systems directly supports class inheritance and bi-directional relationships instead of encoding uni-directional relationships in Foreign Keys or DBRefs.

This was the vision presented by Doug Tolbert in his "Shortcourse on Next Generation Systems" at the Oregon Database Forum in February 1988 (Tolbert).

[5] In this paper, we take the very simple view that RDF is a standard format for storing objects in a triple-store database, OWL is a standard format for storing metadata (i.e., schema information) about the objects in the triple store database, and SPARQL is a standard triple-store query language. More details can be found in the RDF Primer (RDF, 2014), OWL 2 Primer (OWL, 2014), and the SPARQL 1.1 Overview (SPARQL, 2014), however, these references often obscure this simple view.

In his presentation, Doug showed the following diagrams to illustrate his firm conviction that the semantics of the data and operations on the data should be in the database management system (illustrated in the diagram on the right), not in the application program or in the user's mind (illustrated in the diagrams in the middle and on the left).

Today, with Relational systems, we're barely in the middle; with NoSQL databases and Hadoop[6]-like clusters, we're moving to the left. **We need to be moving to the right**!



It is the authors' opinion that this is where the efforts in improving data management should be directed instead of perusing NoSQL database technology with all of its shortcomings in the hope that someday NoSQL database technology will eventual evolve into something better than Relational technology.

Keywords: Data Management, Query Translation, RDF, OWL, SPARQL, SQL, and Inference.

Table of Contents:

1. Introduction

NoSQL database systems, like MongoDB (MongoDB, 2014) and Cassandra (Cassandra, 2014), have been gaining notoriety over the past several years, for instance, there is a large meetup group with thousands of members in New York City devoted solely to NoSQL databases (NoSQL NYC, 2014).

[6] "The Apache Hadoop software library is a framework that allows for the distributed processing of large data sets across clusters of computers using simple programming models." (Hadoop) Hadoop won't be discussed in this paper other than to point out that it represents a move to the far left in Doug Tolbert's representation of data management.

This rapid growth has been due to a few alleged key advantages of NoSQL databases over Relational databases including support for:

1. flexible, “schema-less” data models that are claimed to be better suited for applications developed using the “Agile” methodology (AgileMongoDB, 2014) such as web-based applications.
2. Agile database application development (AgileMongoDB, 2014)
3. RESTful (Tilkov) services in web-based applications and in Database-as-as-Service (DBaaS) (Mongolab, 2014) Cloud environments
4. so called Webscale, “scaleout” database applications

Regarding Item 1 above, the underlying “schema-less” data models for NoSQL databases are strikingly similar to the Entity-Attribute-Value (EVA) data model, which has a rich, decades-long history (Stead, Hammond, & Straube, 1982), (McDonald, Blevins, Tierney, & Martin, 1988). Fortunately, EAV technology has evolved into the standards-based RDF/OWL and SPARQL technology of today and this paper will show that RDF/OWL are practical alternatives to “schema-less” data models found in NoSQL databases.

Item 2 above is intimately tied to Item 1 so, demonstrating that RDF/OWL is a suitable alternative for “schema-less application development, will be sufficient to demonstrate that RDF/OWL (and SPARQL) are suitable for supporting Agile database application development. Therefore, no further discussion of this topic will be found in this paper.

RESTful web-based application development discussed in Item 3 above will be demonstrated in the case study presented in this paper using the RESTful services of Flask (Flask, 2014) (Flask-RESTful, 2014). An implementation of a RESTful server API in ReL for the R language that can be used in a DBaaS, Cloud environment will be discussed in Section 4 of this paper to demonstrate this feature of NoSQL databases.

“Joins” and “Aggregation” using SPARQL will also be demonstrated in the examples found in Section 5 along with an overview of the algorithm used in ReL to translate from SQL to SPARQL.

Item 4 above will not be discussed in this paper because it is believed to be orthogonal to the discussion and it has already been pointed out in the Abstract that existing RDF/Owl implementations provide ACID transaction processing support (Oracle Graph, 2014) in scaleup and scaleout configurations (Exadata, 2014).

Inferencing will be discussed in Section 6 with the emphasis on how it can be a guide to a better future for Data Management.

2. Case Study

A case study looking at the implementation of a Flask-based library website with simple CRUD[7] functionality is used to make the argument that RDF/OWL and SPARQL technology provide an alternative to current NoSQL technologies. Two identical web applications are developed for this study using MongoDB and Cassandra for the NoSQL backends. These applications are then compared to a

[7] Create, Retrieve, Update, Delete.

third implementation developed using RDF/OWL and SPARQL as the backend. In this application, the RDF/OWL and SPARQL database is embedded in an abstraction framework called ReL[8].

Figure 1 shows the main menu for the "library" website example used in this paper. The full code for all three implementation can be found at https://github.com/IsabellaBhardwaj/bookdb/tree/master/.

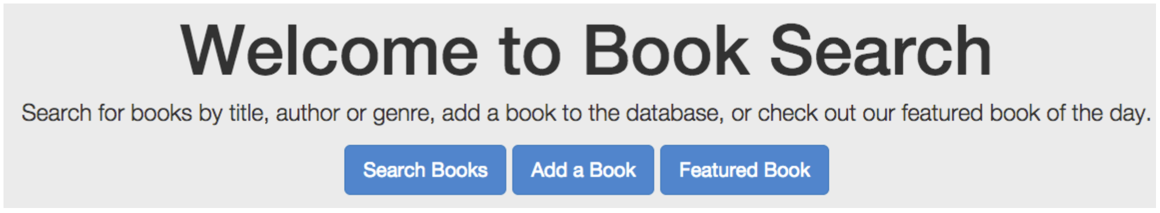

Welcome to Book Search
Search for books by title, author or genre, add a book to the database, or check out our featured book of the day.
Search Books
Add a Book
Featured Book

Figure 1

2.1 Inserting book data into the library website database

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

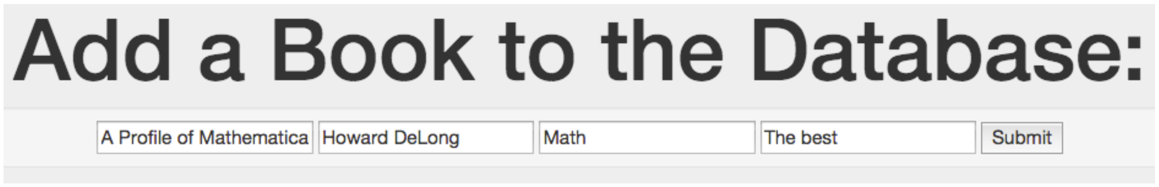


Figure 2

When the submit button is clicked, the code for each implementation as shown in Appendix C is executed. A high level summary of the code follows:

A. <u>For the MongoDB application</u>

In the MongoDB application, the MongoDB "insert" API is called using the following statement.

books.insert(new_data)

[8] ReL (Relation Language) is a python-based, data management system that uses RDF/OWL and SPARQL as its tuple manager. In addition to supporting RDF/OWL and SPARQL, ReL is data model agnostic and 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 data 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 JSON into one of the other supported data models and to translate results back into JSON. This has been done in several ReL applications.

In this statement, “books” is a connection to the database and “new_data” is a python dictionary returned from the web page, which contains the data to be inserted. This is an atomic operation.

B. For the Cassandra application

In the Cassandra application, we use the Cassandra database as a triple-store[9]. The first step in adding the book information to the database is to get a unique identifier to use as the “subject” of the triple. The uuid4() function is used for this. Then the triples are added as follows.

```
id = uuid.uuid4()
insert_statement = "INSERT INTO "+table_name+"(id,
                              property, value) values("+str(id)+", %s, %s)"
batch = BatchStatement()
batch.add(insert_statement, ('title', new_data['title']))
batch.add (insert_statement, ('author', new_data['author']))
batch.add (insert_statement, ('genre', new_data['genre']))
batch.add (insert_statement, ('description', new_data['description']))
session.execute(batch
```

In these statements, “session” is a connection to the database and “new_data” is a python dictionary returned from the web page, which contains the data to be inserted. This needs to be changed by adding a “Batch” statement which is ACD, however “But there is one failure scenario that the classic batch design does not address: if the coordinator itself fails mid-batch, you could end up with partially applied batches.”, see http://www.datastax.com/dev/blog/atomic-batches-in-cassandra-1-2. Also “Although an atomic batch guarantees that if any part of the batch succeeds, all of it will, no other transactional enforcement is done at the batch level. For example, there is no batch isolation. Clients are able to read the first updated rows from the batch, while other rows are still being updated on the server. However, transactional row updates within a single row are isolated: a partial row update cannot be read.”, see http://www.datastax.com/documentation/cql/3.0/cql/cql_reference/batch_r.html

C. For the ReL application

In the ReL application, a “values” variable is set equal to a python tuple of the data values that are to be inserted for a book. Once again, “new_data” is a python dictionary returned from the web page, which contains the data to be inserted.

```
values = (str(new_data['title']), str(new_data['author']),
                      str(new_data['genre']), str(new_data['description']))
```

Then, the data is inserted using standard SQL insert syntax, however, no table named “books” was created beforehand, so this is “scheme-less,” and “flexible” just like MongoDB and Cassandra:

```
SQL on conn "insert into books(title, author, genre, description)
                                                           values"values
```

[9] This is done to easily demonstrate the Cassandra feature where “Unlike a table in an RDBMS, different rows in the same column family do not have to share the same set of columns, and a column may be added to one or multiple rows at any time” (Cassandra, 2014), which is claimed to be a large part of the “Agile” nature of NoSQL databases.

In ReL, the full SQL statement that is passed to the interpreter is a concatenation of the python string beginning with “insert” and the python variable “values”. “Conn is a python variable that holds a connection to the Oracle RDF triple-store database.

ReL converts the SQL statement into a series of insert statements similar the session.execute statements seen in the previous Cassandra example. These statements are enclosed in a standard Oracle ACID transaction. The following is a subset of the statements generated but ReL; the complete set of statements can be found in Appendix C.

The first three statements below define the Oracle transaction and the fourth statement inserts the data for the “title” attribute into the RDF triple-store.

```
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'));
```

The next four **INSERT** statements insert meta-data (i.e., schema information) for the “title” attribute into the RDF triple-store. The four statements say the “title” is of type “owl:DatatypeProperty”, its domain is “books”, its range is “string”, and it’s of type “owl:FunctionalProperty”. “BOOK_APP_SQNC.nextval” is a GUID similar the id in the Cassandra example.

```
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 library database.

2.2 Searching the book website database

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

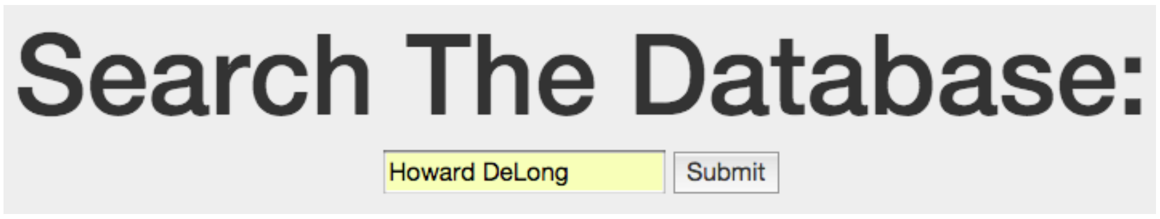


Figure 3

When the submit button is clicked, the code for each implementation as shown in Appendix D is executed. A high level summary of the code follows:

A. For the MongoDB application

In the MongoDB application, the MongoDB "find" API is called using the following statement.

```
books.find({'$or':[{'title':query},{'author':query}]})
```

B. For the Cassandra applicaiton

```
for row in title_ids:
   id = row.id
   title_name = session.execute(value_select_statement, (id,'title'))[0]
   author_name = session.execute(value_select_statement, (id, 'author'))[0]
   inner_dict = {'title': title_name.value, 'author': author_name.value}
   result_dict[str(id)] = inner_dict

for row in author_ids:
   id = row.id
   title_name = session.execute(value_select_statement, (id, 'title'))[0]
   author_name = session.execute(value_select_statement, (id, 'author'))[0]
   inner_dict = {'title': title_name.value, 'author': author_name.value}
   result_dict[str(id)] = inner_dict
```

C. 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):

```
query = request.form['query']

titles[10] = 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 = convert_to_dict(titles)
author_dict = convert_to_dict(authors)
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
```

[10] The SQL statement returns a python tuple of tuples.

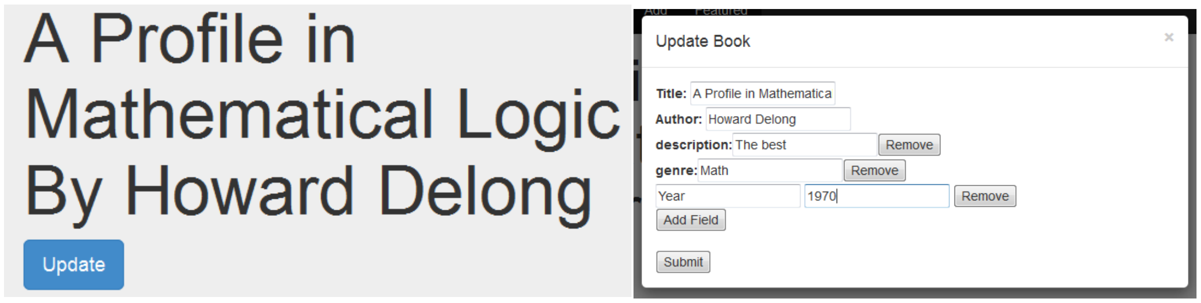
Behind the scenes, ReL converts the SQL select statements into SPARQL statements, as shown below[11,12]:

```
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) )
```

2.3 Updating the book website database

Figure 4 shows the menus for updating book information in this library application.


A Profile in Mathematical Logic By Howard Delong

Update

Update Book

Title: A Profile in Mathematica

Author: Howard Delong

description: The best Remove

genre: Math Remove

Year 1970 Remove

Add Field

Submit

Figure 4

When the submit button is clicked, the code for each implementation as shown in Appendix D is executed. A high level summary of the code follows:

A. **For the MongoDB application**

[11] Notice, the SPARQL statement is in an Oracle Table function that is a part of a standard SQL statement.

[12] The use of the OPTIONAL pattern in the SPARQL statements means that each of the attributes modified by the OPTIONAL pattern will optionally be part of a returned tuple (row). This is like the CASSANDRA NoSQL database system where "Unlike a table in an RDBMS, different rows in the same column family do not have to share the same set of columns, and a column may be added to one or multiple rows at any time." (Cassandra, 2014)

B. **For the Cassandra application**

C. **For the ReL application**

3. Transaction Discussion

In the case study, it was shown that Oracle's RDF implementation (Oracle Graph, 2014), has support for standard transaction properties such as read consistency, serializable (Concurrency, 2011), and ACID (Transactions, 2011) so these features could be made available to NoSQL-like applications. As stated earlier in the paper, this addresses a major weakness of NoSQL databases; that they do not provide this level of transaction support. MongoDB, for example, only supports single statement transactions and the entire Database[13] is locked by the transaction statement and Cassandra …

The SALT project at the University of Texas summarizes the situation as, "The ACID vs. BASE debate is well known. In one corner are ACID transactions: through their guarantees of Atomicity, Consistency, Isolation, and Durability, they offer an elegant and powerful abstraction for structuring applications and reasoning about concurrency, while ensuring the consistency of the database despite failures. Such ease of programming, however, [sometimes]comes at a significant cost of performance and availability. In the other corner is the BASE approach, recently popularized by several NoSQL systems. BASE avoids distributed transactions to eliminate the performance and availability costs of the associated distributed commit protocol. Embracing the BASE paradigm, however, exacts its own heavy price: once one renounces ACID guarantees, **it is up to developers to explicitly code in their applications the logic necessary to ensure consistency in the presence of concurrency and faults, and the complexity of this task easily gets out of control**."

This is complete opposition to the tenant that "the semantics of the data and operations on the data should be in the database management system" envisioned by Doug Tolbert in his "Shortcourse on Next Generation Systems" at the Oregon Database Forum in February 1988 (Tolbert).

4. RESTful Discussion

The case study demonstrated how RESTful, web-based application development can be done in MongoDB, Cassandra, and ReL using the RESTful services of Flask (Flask, 2014) (Flask-RESTful, 2014).

Similarly, the RESTful support facilities found at MongoLab (Mongolab, 2014) and Cassandra's CloudCredo (CloudCredo) to support a DBaaS environment are also available in ReL. As an example, an implementation of a RESTful server API in ReL for the R language can be embedded in any environment that supports the CURL function. For instance, we use RESTful ReL in R (Project, 2014)

[13] A Mongo Database is a container for a set of Collections and a Collection is a container for a set of JSON Documents.

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 RESTful 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
```

Therefore, all of the RDF/OWL and SPARQL technology, along with transaction support can be made available in a DBaaS, Cloud environment, e.g., (Oracle Public Cloud)[14].

[14] Dr. Cannata and his students are implementing a DBasS Cloud environment for his Data Science classes at the University of Texas on Oracle’s Public Cloud using the RESTful ReL technology

5. SQL to SPARQL Discussion and Examples

Here is an overview of the SQL to SPARQL algorithm. MORE WORK REQUIRED FOR THIS SECTION

```
SQL:
"SELECT p1, p2, ... p_p
FROM t1 join t2 on (equality)
WHERE f_1 (expression) AND f_2 (expression)...
GROUP BY g_1, g_2, ... g_g
HAVING h_1 (expression) AND h_2 (expression) AND ... h_h (expression)
ORDER BY o_1, o_2, ... o_o"

SPARQL:
"SELECT v1 "REGULAR", n1 "AGGR", ... v_p "P_MAX"
 FROM TABLE(SEM_MATCH('SELECT ?v1 (avg(?v2) as ?n1) ...  ?_v "P_MAX"
 WHERE {?s1 rdf:type :TABLE_1.
...
# use ?s1 for the rest of example
OPTIONAL { ?s1 :SELECT_1 ?v1 }
OPTIONAL { ?s1 :SELECT_2 ?v2 }
...
OPTIONAL { ?s1 :SELECT_MAX ?v_(p+o+g+h}
?s1 :FILTER_1 ?f1
FILTER(?f1 [expression])
}
GROUP BY ?v_(p+o+1) ... ?v_(p+o+g)
HAVING(?v_(p+o+g+1) [expression] ... ?v_(p+o+g+h)
ORDER BY ?v_(p+1) ... ?v_(p+o)
```

table t (in order of appearance in SQL query) is mapped to variable ?s_t:
?s_t rdf:type :TABLENAME

project columns p1 through p_p are converted to ?v1 to ?v_p
orderby columns o_1 to o_o are converted to ?v_(p+1) to ?v_(p+o)
groupby columns g_1 to g_g are converted to ?v_(p+o+1 to ?v_(p+o+g)
having columns h_1 to h_h are converted to ?v_(p+o+g+1) to ?v_(p+o+g+h)

For each column variable ?v_i, an optional will map the column name to it:
OPTIONAL {?s_tableno :colName ?v_i}

For each optional select, the columns which will be projected are put in the SELECT clause:
SELECT ?v1 ?v2 ... ?v_p
aggregates are selected in a new indexing system ?n1... as such:
(avg(?v_aggr_1) as ?n1

For each variable to be projected, we get their alias.

---

discussed in this paper. A large set of publically available data will be hosted in this environment for visualization and data mining analysis.

SELECT v1 "COLUMN ALIAS", n1 "AGGR COLUMN ALIAS"

Append filters after the optional selects:
?s_tableno :colname ?f_filterno
...
FILTER(?f_filterno [expression])
...
}

Now look through stored group by, having, and order by variables, and append them to the SPARQL.
The minimum i value for any ?v_i here is going to be p + 1 (number of projected columns + 1)
GROUP BY ?v_g
HAVING(?v_h)
ORDER BY ?v_o

5.1 Basic Example

SQL:
```
select deptno, sal from emp where SAL > 1000
```

SPARQL:

```
SELECT v1 "DEPTNO", v2 "SAL" /* Name them */
 FROM TABLE(SEM_MATCH('SELECT ?v1 ?v2 WHERE { /* Project them */
?s1 rdf:type :EMP .
OPTIONAL { ?s1 :DEPTNO ?v1 } /* Select them */
OPTIONAL { ?s1 :SAL ?v2 }
OPTIONAL { ?s1 :SAL ?v3 }
?s1 :SAL ?f1 .
FILTER(?f1 > 1000)
}
```

5.2 Join Example[15]

SQL:
```
select deptno, sal from emp e, dept d where e.deptno = d.deptno
and SAL > 1000
```

SPARQL:
```
TBD
```

5.3 Complete Example

SQL:

```
select deptno, avg(sal) from emp
group by deptno
having avg(sal) > 1000
order by avg(sal)
```

[15] The standard Oracle scott/tiger schema is used in these examples (Scott Schema)

SPARQL:

```
SELECT v1 "DEPTNO", n2 "AVG(EMP.SAL)"
 FROM TABLE(SEM_MATCH('SELECT ?v1 (avg(?v2) as ?n1) WHERE {
?s1 rdf:type :EMP .
OPTIONAL { ?s1 :DEPTNO ?v1 }
OPTIONAL { ?s1 :SAL ?v2 }
OPTIONAL { ?s1 :SAL ?v3 }
OPTIONAL { ?s1 :SAL ?v4 }
OPTIONAL { ?s1 :DEPTNO ?v5 }
}
GROUP BY ?v5
HAVING( avg(?v3) > 1000)
ORDER BY avg(?v4)
```

6. Inferencing Discussion

OWL offers a large number of inference rules that can be added to a triple store from which new data can be entailed. For example,

7. 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 “schema-less”, flexible NoSQL-type applications and DBaaS applications. What’s more, unlike other NoSQL systems, RDF/OWL and SPARQL systems like ReL can support standard, read committed, serializable, and ACID transaction processing. 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 similar implementation of these standards could be used instead.

Inference has the potential to be a very powerful technology in data management, especially the notions of “type” and “inverse” . . .

These ideas, when added to a DBMS, will revolutionize the data management field. There will no longer be a need to create a conceptual model where class hierarchies are modeled which then need to be converted to a logical model where the hierarch is collapsed in on of four way which then has to be transformed to a relational mode where relationships are transformed to. The conceptual model will be the dbms’s schema whether explicit or implicit.

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 a matter of fact, ReL and RESTful ReL would run blazingly fast in Oracle’s Exadata (Exadata, 2014) environment, which is effectively “scale-out” albeit not inexpensive.

8. Bibliography

AgileMongoDB. (2014). Retrieved from http://www.mongodb.com/agile-development

Brewer, E. (2000). *Keynote Address at Nineteenth ACM Symposium on Principles of Distributed Computing (PODC 2000)*. Retrieved from http://www.cs.berkeley.edu/~brewer/cs262b-2004/PODC-keynote.pdf

Cassandra. (2014). Retrieved from http://en.wikipedia.org/wiki/Apache_Cassandra

Chao Xie, C. S. (2014). *Salt: Combining ACID and BASE in a Distributed Database*. Retrieved from https://www.usenix.org/conference/osdi14/technical-sessions/presentation/xie

Concurrency. (2011). Retrieved from http://docs.oracle.com/cd/E28271_01/server.1111/e25789/consist.htm

Exadata, O. (2014). Retrieved from https://www.oracle.com/engineered-systems/exadata/index.html

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/

NoSQL NYC. (2014). Retrieved from http://www.meetup.com/nosql-nyc/?gj=ej1c&a=wg2.1_recgrp

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

OWL. (2014). Retrieved from http://www.w3.org/TR/owl2-primer/
Project, R. (2014). Retrieved from http://www.r-project.org/

RDF. (2014). Retrieved from http://www.w3.org/TR/2014/NOTE-rdf11-primer-20140225/

Seth Gilbert, N. L. (2002). *Brewer's Conjeture and the Feasibility of Consistent, Available, Partition-Tolerant Web Servies*. Retrieved from http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.20.1495&rep=rep1&type=pdf

SPARQL. (2014). Retrieved from http://www.w3.org/TR/sparql11-overview/
SQL Server Scale Out White Paper - Microsoft. (n.d.). Retrieved from http://download.microsoft.com/download/5/B/E/5BEFA55A-19B0-43EA-BEA4-6E8E7641E163/SQL_Server_2012_Scale_Out_White_Paper_Mar2012.docx

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.

Transactions. (2011). Retrieved from http://docs.oracle.com/cd/E28271_01/server.1111/e25789/transact.htm#g11401

Appendix A - Creating the Databases

TBD

Appendix B – Connecting to the Databases

Making a MongoDB connection

```
import pymongo

connection_string = "mongodb://127.0.0.1"
connection = pymongo.MongoClient(connection_string)
database = connection.books
books = database.mybooks
```

Making a Cassandra connection and create the triple-store table if it doesn't already exist

```
from cassandra.cluster import Cluster
import uuid

cluster = Cluster()
session = cluster.connect('keyspace1')

table_name = "new_table"
session.execute("CREATE TABLE IF NOT EXISTS %s(id uuid, property text, value text, primary key(id, property));"%table_name)
session.execute("CREATE INDEX IF NOT EXISTS on %s(value);"%table_name)
```

Making a ReL connection

```
conn = connectTo 'jdbc:oracle:thin:@host:1521:orcl' 'user' 'password' 'rdf_mode' 'rdf_model'
```
[16]

[16] Creating a connection also creates an RDF Model and some utility sequences if the Model doesn't already exit as follows,

```
EXECUTE IMMEDIATE 'CREATE TABLE F2014_C##CS347_PROF_DATA( id NUMBER, triple SDO_RDF_TRIPLE_S)';
SEM_APIS.CREATE_RDF_MODEL('F2014_C##CS347_PROF', 'F2014_C##CS347_PROF_DATA', 'triple');
EXECUTE IMMEDIATE 'CREATE SEQUENCE F2014_C##CS347_PROF_SQNC MINVALUE 1 START WITH 1 INCREMENT BY 1 NOCACHE';
EXECUTE IMMEDIATE 'CREATE SEQUENCE F2014_C##CS347_PROF_GUID_SQNC MINVALUE 1 START WITH 1 INCREMENT BY 1 NOCACHE';
```

Appendix C – Adding a book to the library

MongoDB Code

```
@app.route('/add/', methods=['GET', 'POST'])
def add():
      if request.method == 'POST':
            new_data = {k : v for k, v in request.form.items()}
            #If the user leaves a field blank
            if new_data['title'] == '' or new_data['author'] == '' or
               new_data['genre'] == '' or new_data['description'] == '':
                  return render_template('add.html', alert="required")
            #If the user tries to add a book that's already in the database
            elif books.find({'title':new_data['title'],
                  'author':new_data['author']}).count() > 0:
                  return render_template('add.html', alert="exists")
            else:
                  books.insert(new_data)
                  return render_template('add.html', alert = "success")
      else:
            return render_template('add.html', alert="")
```

Cassandra Code:

```
@app.route('/add/', methods=['GET', 'POST'])
def add():
      if request.method == 'POST':
            new_data = {k : v for k, v in request.form.items()}
            #If the user leaves a field blank
            if new_data['title'] == '' or new_data['author'] == '' or
               new_data['genre'] == '' or new_data['description'] == '':
                  return render_template('add.html', alert="required")
            else:

                  id = uuid.uuid4()
                  batch = BatchStatement()
                  insert_statement = "INSERT INTO "+table_name+"(id,
                                   property, value) values("+str(id)+", %s, %s)"
                  batch.add(insert_statement, ('title',
                                                          new_data['title']))
                  batch.add (insert_statement, ('author',
                                                         new_data['author']))
                  batch.add (insert_statement, ('genre',
                                                          new_data['genre']))
                  batch.add (insert_statement, ('description',
                                                     new_data['description']))
                  session.execute(batch)

                  return render_template('add.html', alert = "success")
      else:
            return render_template('add.html', alert="")
```

ReL Code:

```
@app.route('/add/', methods=['GET', 'POST'])
def add():
    if request.method == 'POST':
```

```
    new_data = {k : v for k, v in request.form.items()}
    #If the user leaves a field blank
     if new_data['title'] == '' or new_data['author'] == '' or
       new_data['genre'] == '' or new_data['description'] == '':
        return render_template('add.html', alert="required")

    #If the user tries to add a book that's already in the database
    # elif books.find({'title':new_data['title'],
                             'author':new_data['author']}).count() > 0:
      # return render_template('add.html', alert="exists")
    else:
      # books.insert(new_data)
      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 follows[17] (data level triples are shown in bold below and OWL level triples are shown in italics and underlined).

```
BEGIN
commit ;
set transaction isolation level serializable[18] ;
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'));
```

[17] URIs have been abbreviated to help with readability.
[18] Notice that Oracle allows RDF triple-store statements to be wrapped in standard SQL. This means standard transaction processing can be done with RDF triple-store statements.

```
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 ;
/
```

Appendix D – Searching for books in the library

MongoDB Code

```
@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']
		result_cursor =
		          books.find({'$or':[{'title':query},{'author':query}]})
		no_results = result_cursor.count() == 0
		result_dict = convert_to_dict(result_cursor)
		return render_template('search.html', posting=True,
		        query=query, no_results=no_results, results=result_dict)
	else:
		return render_template('search.html', posting=False)
```

Cassandra Code

```
@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']

		id_select_statement = "SELECT id FROM "+table_name+" WHERE
		                   property = %s and value = %s ALLOW FILTERING"
		title_ids = session.execute(id_select_statement, ('title',query))
		author_ids = session.execute(id_select_statement, ('author',
		                                                         query))
		value_select_statement = "SELECT value FROM "+table_name+" WHERE
		              id = %s and property = %s LIMIT 1 ALLOW FILTERING"
		result_dict = {}
		for row in title_ids:
			id = row.id
			title_name = session.execute(value_select_statement, (id,
			                                            'title'))[0]
			author_name = session.execute(value_select_statement, (id,
			                                           'author'))[0]
			inner_dict = {'title': title_name.value, 'author':
			                                       author_name.value}
			result_dict[str(id)] = inner_dict

		for row in author_ids:
			id = row.id
			title_name = session.execute(value_select_statement, (id,
			                                            'title'))[0]
			author_name = session.execute(value_select_statement, (id,
			                                           'author'))[0]
			inner_dict = {'title': title_name.value, 'author':
```

```
                                                                  author_name.value}
                result_dict[str(id)] = inner_dict
           return render_template('search_cass.html', posting=True,
                                            query=query, results=result_dict)
      else:
           return render_template('search_cass.html', posting=False)
```

ReL Code

```
@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
        # title_dict = {}
        # author_dict = {}
        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)
```

Appendix E – Updating book information in the library

MongoDB Code

```
@app.route('/detail/<title>/<author>/', methods=['GET', 'POST'])
def detail(title, author):
	if request.method == 'GET':
		cursor = books.find_one({'title':title, 'author':author})

	elif request.method == 'POST':
		#Add new values of all pre-existing attributes
		updated_document = {attribute: value for attribute, value in
request.form.iteritems() if attribute[:9] != 'new_field' and attribute[:9]
!= 'new_value'}
		num_old_fields = len(updated_document)
		num_new_fields = (len(request.form)-num_old_fields)/2
		#Add values of new fields, if any
		if(num_new_fields > 0):
			for i in range(1, num_new_fields + 1):
				new_attribute = request.form['new_field'+str(i)]
				new_value = request.form['new_value'+str(i)]
				updated_document[new_attribute] = new_value
		books.update({'title':title, 'author': author}, updated_document)
		cursor = books.find_one({'title': request.form['title'],
'author': request.form['author']})

	results = {field:value for field, value in cursor.items()}
	return render_template('detail.html', result=results)
```

Cassandra Code

```
@app.route('/detail/<id>/', methods=['GET', 'POST'])
def detail(id):
	id = uuid.UUID(id)
	if request.method == 'POST':
		#In the dict request.form, pre-existing properties and values
make up key-value pairs, with the property being the key and the value being
the value. New properties and values are all values in the dictionary, and
their keys are named "new_field"+str(pair_number) and
"new_value"+str(pair_number), respectively. pair_number is a digit that
identifies which new property goes with which new value.
		for key, value in request.form.iteritems():
			#add new property and value to book
			if key[:9] == 'new_field':
				pair_number = key[9:]
				session.execute("UPDATE "+table_name+" SET value = %s
WHERE id = %s and property =
%s",(request.form['new_value'+str(pair_number)], id, value))
			#update value of existing property of book
			elif key[:9] != 'new_value':
				session.execute("UPDATE "+table_name+" SET value = %s
WHERE id = %s and property = %s",(value, id, key))
```

```
	result = {}
	all_properties = session.execute("SELECT property, value FROM 
"+table_name+" WHERE id = %s", (id,))
	for property in all_properties:
		result[property.property] = property.value
	return render_template('detail_cass.html', result=result, id=id)
```

ReL Code