RDF/OWL and SPARQL instead of NoSQL databases

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

ABSTRACT

A few shortcomings in popular NoSQL implementations make widespread adoption of this schema-less database technology difficult in many application domains. Such shortcomings include a lack of a standard query language, no in-database support for joins, and no support for full ACID transaction processing.

This paper proposes that RDF/OWL and SPARQL implementations provide an alternative to NoSQL implementations without these shortcomings, while still providing the advantages of existing NoSQL implementations, including being flexible and schema-less, providing support for agile application development, and working well in RESTful [Tilkov] service and Database-as-a-Service (DBaaS) [*Mongolab* 2014; *CloudCredo*] environments. This paper also outlines the added promise of Inference as part of a data management system, which is an existing feature in RDF/OWL-based implementations.

Keywords

Data Management, Query Translation, RDF, OWL, SPARQL, SQL, NoSQL, ReL, SIM, and Inference

1. INTRODUCTION

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. The following is a list of these shortcomings:

1. Lack of a standard query language.
2. No support for joins, thus, encouraging applications to de-normalize their data.[2]
3. Limited support for ACID [Transactions 2011] transaction processing[3] as found in modern data management systems because of a perceived need for "inexpensive" scale out.[4]
4. A creep of proprietary features, such as MongoDB aggregation [MongoDB Aggregation 2014].

These shortcomings make widespread adoption of NoSQL datastores impractical and difficult. The lack of standard a query language and creep of proprietary features ties developers to unique datastores with little opportunity to replace the backend database if required. This is undesirable from a business perspective. The lack of join support reduces the application domain of NoSQL technology significantly, unless developers are willing to engage in considerable data mashup in their code. Also, without support for full ACID transaction processing, true asynchronous critical applications are impossible.

---

[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 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 DISCUSSION," for more details.

[4] "Scalability is the ability of an application to efficiently use more resources in order to do more useful work...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: scale up and scale out. Scale up 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... Scale up has the advantage of not requiring significant changes to the database...Scale out means expanding to multiple servers rather than a single, bigger server. Scale out 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 scale-out solution is also useful from an availability perspective" [SQL Server Scale Out White Paper – Microsoft].

As an alternative, this paper proposes that RDF/OWL and SPARQL database technology[5] fills the void where current NoSQL implementations have failed. Below is a list of highlights why:

1. SPARQL provides a full-featured standard query language. It supports all of the features of an ad hoc query language, such as projections of attributes, selection of instances, aggregation, and sub-queries. This paper demonstrates that SPARQL is very flexible and can act as a lingua franca, allowing other languages, such as SQL, to be used on top of it.
2. RDF/OWL and SPARQL systems support "joins" in the database (see Section 4).
3. SPARQL has full support for features like aggregation (see Section 4).
4. Existing RDF/Owl implementations provide ACID transaction processing support [*Oracle Graph* 2014] in scale-up and scale-out configurations [Exadata 2014].

This paper also shows that the highly touted features of NoSQL technology, listed below, are equally achievable in RDF/OWL and SPARQL based systems making RDF/OWL and SPARQL based systems a powerful alternative to existing NoSQL implementations:

- A flexible, schema-less data model "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].
- Support for agile database application development, which "includes a set of software development methods focused on an iterative approach to building software".[6]
- Works well in RESTful [Tilkov] services in web-based applications and in Database-as-a-Service (DBaaS) [*Mongolab* 2014; *CloudCredo*] environments.

RESTful web-based application development mentioned above is demonstrated in Section 2, "CASE STUDY," using the RESTful services of Flask [*Flask* 2014; *Flask-RESTful* 2014]. Also, an implementation of a RESTful server API for the R language is shown to prove that RDF/OWL technology can be used in a DBaaS, Cloud environment. This is discussed in Section 5, "RESTful DISCUSSION".

Lastly, this paper touches on the potential of the power of Inference [*OWL Inference*] integration, which is an existing feature in RDF/OWL based systems, into data management systems. Inference is a means of deriving logical conclusions from facts already known in the system. The following example shows this simple device:

1. Fact: John has parent Sally.
2. Inference: Sally has child John.

The inference in this example is known as "InverseOf" and can be used to assure the bi-directionality of a relationship (see Section 6, "INFERENCE DISCUSSION," for more details).

Here is another example:

1. Fact: Bob is a member of the male class.
2. Fact: The male class is a subset of the animal class.
3. Inference: Bob is also of the animal class.

This is an important relationship for the support of class inheritance (see Section 6 for more details).

As previously mentioned, Inference is natively supported in OWL-based systems but is not available in Relational or current NoSQL implementations. The capabilities of Inference can, at a minimum, act as a guide to a better future for data management where the database management systems directly support class inheritance and bi-directional relationships instead of encoding uni-directional relationships in Foreign Keys or DBRefs [MongoDBRefs].

This was the vision presented by Doug Tolbert in his lecture, "Shortcourse on Next Generation Systems," at the Oregon Database Forum in February 1988 [Tolbert].

---

[5] In this paper, we take the 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.

[6] 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].

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

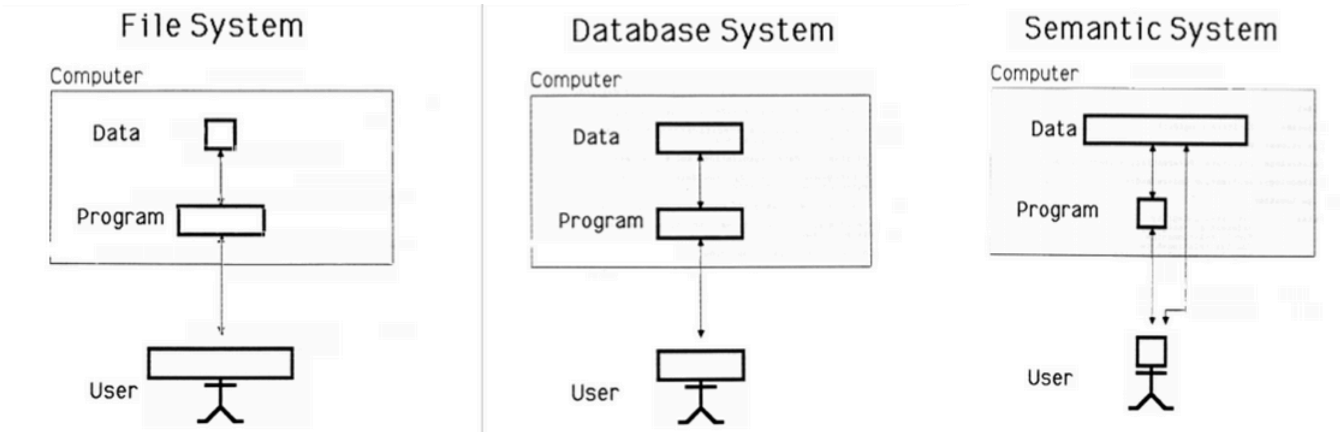

Figure 1

Having the semantics of the data and operations on the data in the database management system is critically important because the system can then provide *common* solutions to important problems, such as modeling complex data and relationships, data integrity, data retrieval, performance, and ease of use. This leaves the application to deal solely with application-level functionality.

Today, with relational systems, we are barely in the middle: with NoSQL databases and Hadoop-like clusters moving to the left putting more weight on the application developers rather than letting the data management system do the heavy lifting. **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 in the hope that someday NoSQL database technology will evolve into something more accessible than relational technology.

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 equivalent 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 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] Close consideration is given to ACID transaction processing in each of these implementations.

Figure 1 shows the main menu for the library website example used in this paper. The full code for all three implementations can be found at https://github.com/IsabellaBhardwaj/bookdb/tree/master/. The ReL version of the application is running at http://129.152.144.84:5000/

[7] Create, Retrieve, Update, Delete.

[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 and 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 is 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.

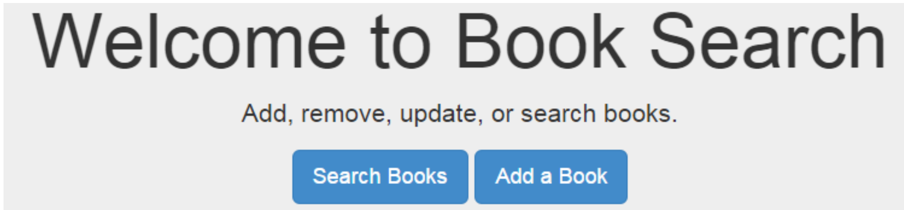

Figure 2

2.1 Inserting Book Data into the Library Website Database

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

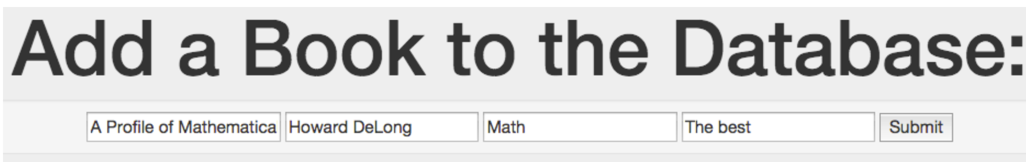

Figure 3

The code for adding a book for each implementation is shown in Appendix C. The following subsections provide a high-level summary of the code.

2.1.1 For the MongoDB Application

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

books.insert(new_data)

In this statement, "books" is a connection to the MongoDB "books" database and "new_data" is a python dictionary returned from the web page, which contains the data to be inserted. This statement has the properties of an ACID transaction [Transactions 2011]; however, the entire "books" database is locked for the duration of the statement. This lock blocks all other connections from reading or writing the document [*Mongo lock granularity*].

"In MongoDB 2.2, only individual operations are **Atomic.** By having per database locks control reads and writes to collections, write operations on collections are **Consistent and Isolated.** With journaling on, operations may be made **Durable**. Put these properties together, and you have basic **ACID** properties for **transactions.**

The shortcoming with MongoDB's implementation is that these semantics apply to individual write operations, such as an individual insert or individual update. If a MongoDB statement updates 10 rows, and something goes wrong with the fifth row, then the statement will finish execution with four rows updated and six rows not updated" [*MongoDB Transactions*; *MongoDB ACID*].

2.1.2 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.[10] The uuid4() function is used for this. Then the triples are added in a "batch" statement [*Cassandra Batch*] as follows:

```
id = uuid.uuid4()
```

[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. What this means in this particular instance is that different books represented by IDs can all have different sets of attributes and values represented by different rows.

[10] The use of a unique identifier in the Cassandra and ReL applications is not natural and, therefore, not done in the MongoDB application.

```
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.

The Cassandra "batch" statements "guarantee 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" [*Cassandra Batch*].

"However, 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." [*Cassandra Atomic*]. "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" [*Cassandra Isolation*].

Brief memo about why this isolation support is so dangerous is required.

2.1.3 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
```

In ReL, the full SQL statement is a concatenation of the python string beginning with "insert" and the interpreters evaluation of the python "values" variable. "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 to the batch.add statements seen in the previous Cassandra example. These statements are enclosed in a standard Oracle ACID transaction, which, unlike Cassandra, provide complete atomicity and isolation with no failure scenarios.

The following is a subset of the statements generated by 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 metadata (i.e., schema information) for the "title" attribute into the RDF triple-store:

```
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'))
```

In these four statements, “title” has the following properties:

- Type = “owl:DatatypeProperty”
- Domain = “books”
- Range = “string”
- Type “owl:FunctionalProperty”

“BOOK_APP_SQNC.nextval” is a GUID similar to the ID in the Cassandra example.

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 4

The code for searching the database for each implementation is shown in Appendix D. The following subsections provide a high-level summary of the code.

2.2.1 For the MongoDB Application

In the MongoDB application, the MongoDB “find” API is called using the following statement, where “query” is a string containing the terms the user typed into the search bar:

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

2.2.2 For the Cassandra Application

In the Cassandra application, the first step is to get the subject uuids of the triples that satisfy the query as follows:

```
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))
```

Next, a result dictionary containing the title and author associated with each subject uuid is constructed as follows:

```
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
```

2.2.3 For the ReL Application

The ReL search is done using standard SQL select statements as follows:

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

         no_results = result_dict == 0
         return render_template('search.html', posting=True, query=query,
                                  no_results=no_results, results=result_dict)
```

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 .
```

---

[11] 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 similar to 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].

```
      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

To update the information about an existing book in the database, enter the title or author into the search bar.

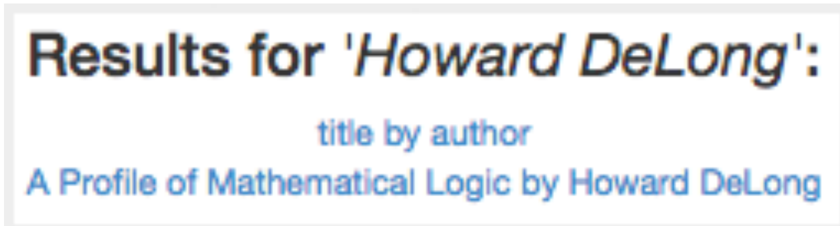

Figure 5

Next, click the link for the book.

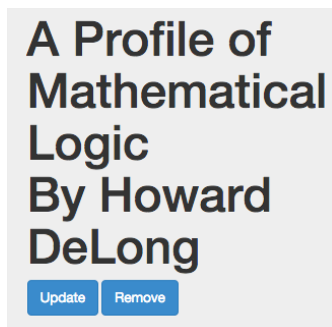

Figure 6

Then, click Update.

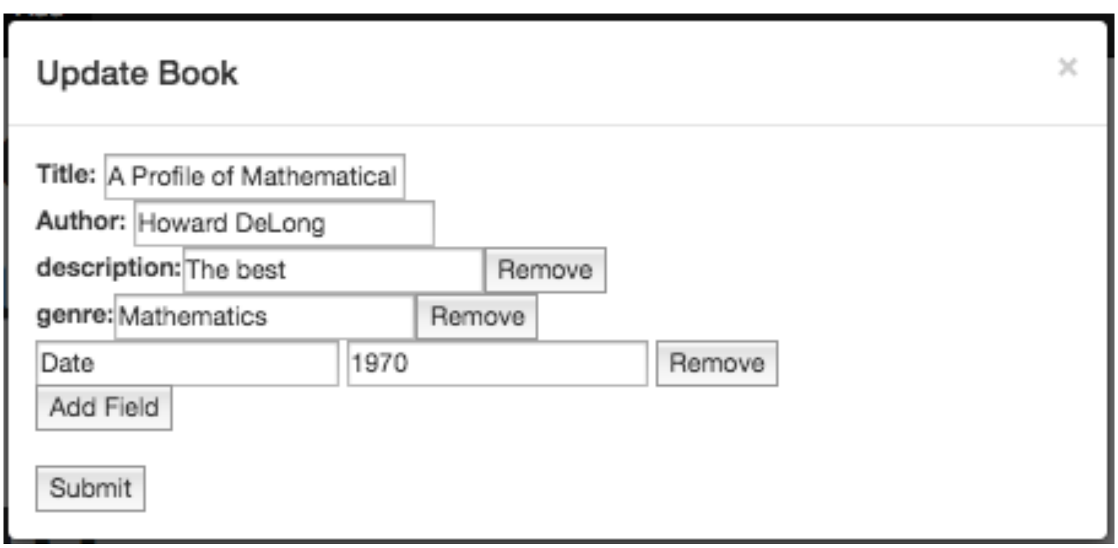

Figure 7

Edit the information. In this case, Math was changed to Mathematics, and a Date field was added. Click the Submit button.

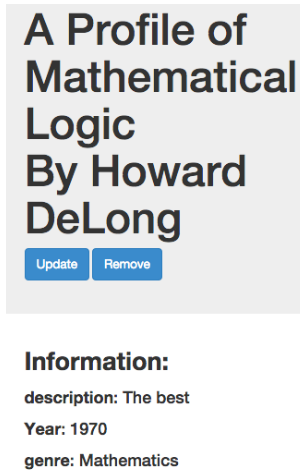

Figure 8

The updated information is displayed.

The code for updating the database for each implementation is shown in Appendix D. The following subsections present a high-level summary of the code.

2.3.1 For the MongoDB Application

In the MongoDB application, the MongoDB "find_and_modify" API is called using the following statement, where "updated_document" is a new document that has been constructed by the application to hold all of the changes:

```
return books.find_and_modify({'title':title, 'author': author},
                                                 updated_document, new=True)
```

By using "find_and_modify" the MongoDB application displays only the changes the current user made to a book, without displaying any changes made at the same time by any other user. However, it should be pointed out that "find_and_modify" does not work in many cases because it can update and return only one document, and you cannot specify the write concern of the operation when using it.

The alternative to using "find_and_modify" is to use update(). update() can update more than one document, and the user can specify the write concern. The problem with using update() in this application is that it does not return the modified document. As a result, after using update(), a separate books.find() query is necessary to retrieve the modified document in order to display the book's updated information on the webpage. If a second user made changes to the document in between the time that the first user's update() and find() queries were executed, the changes that the second user made would be displayed to the first user, which is not ideal.

2.3.2 For the Cassandra Application

In the Cassandra application, the following statement that performs the addition of the "Year" field is prepared for "batch" mode[13] execution:

```
batch.add("UPDATE "+table_name+" SET value = %s WHERE id = %s and property =
```

[13] "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" [*Cassandra Batch*].

```
            %s",(request.form['__new__value__'+str(pair_number)], id, value))
```

Next, the following statement that performs the update, changing the value of “genre” field from “Math” to “Mathematics”, is prepared for “batch” mode execution:

```
 batch.add("UPDATE "+table_name+" SET value = %s WHERE id = %s and property = %s",(value, id,
key))
```

The “batch” is executed:

```
session.execute(batch)
```

Then, the following query returns the updated information to the web application:

```
session.execute("SELECT property, value FROM
                                       "+table_name+" WHERE id = %s", (id,))
```

The select statement cannot be included in the batch statement, so it cannot be a part of the batch transaction. Just like MongoDB, This means that updates from other sessions not just the updates from this session could appear in the final display of the data.

2.3.3 For the ReL Application

The ReL application generates the following sequence of RDF/SPARQL code for the update. First, the following code performs an update, changing the value of” genre” from “Math” to “Mathematics”:

```
UPDATE BOOK_C##CS347_PROF_DATA a
SET a.triple = SDO_RDF_TRIPLE_S ('BOOK_C##CS347_PROF',
     a.triple.get_subject(),
       '<http://www.example.org/people.owl#genre>',
       '"Mathematics"^^<http://www.w3.org/2001/XMLSchema#string>')
WHERE a.triple.get_subject() =
                              '<http://www.example.org/people.owl#8>'
AND a.triple.get_property() =
                          '<http://www.example.org/people.owl#genre>'
AND a.triple.get_obj_value() =
                  '"Math"^^<http://www.w3.org/2001/XMLSchema#string>'
```

Next, the following code performs the addition of the “Year” field:

```
INSERT INTO BOOK_C##CS347_PROF_DATA VALUES (BOOK_C##CS347_PROF_SQNC.nextval,
SDO_RDF_TRIPLE_S('BOOK_C##CS347_PROF:<http://www.example.org/people.owl>',
'http://www.example.org/people.owl#8', 'http://www.example.org/people.owl#Year',
'"1970"^^<http://www.w3.org/2001/XMLSchema#string>'))
```

Schema information for the “Year” field is also inserted in a manner similar to the discussion in Section 2.1.

Lastly, the following query returns the updated information to the web application:

```
SELECT v2 "genre", v3 "Year", v4 "description", v5 "author",
v6 "title"
 FROM TABLE(SEM_MATCH('SELECT * WHERE {
       ?s1 rdf:type :books .
       OPTIONAL { ?s1 :genre ?v2 }
       OPTIONAL { ?s1 :Year ?v3 }
       OPTIONAL { ?s1 :description ?v4 }
       OPTIONAL { ?s1 :author ?v5 }
       OPTIONAL { ?s1 :title ?v6 }
       ?s1 :title ?f1 .
       ?s1 :author ?f2 .
       FILTER(?f1 = "A Profile of Mathematical Logic" && ?f2 = "Howard DeLong") }' ,
SEM_MODELS('BOOK_C##CS347_PROF'), null,
SEM_ALIASES( SEM_ALIAS('', 'http://www.example.org/people.owl#')), null) )
```

Unlike the MongoDB and Cassandra updates, all three of these operations can be enclosed in a standard Oracle ACID transaction.

3 TRANSACTION DISCUSSION

In the case study, we showed the following:

1. MongoDB:
   a. Only single statement operations on a document can have ACID transaction properties.
   b. A statement that writes to the database locks the entire database for the duration of the statement.
   c. The database lock blocks all other connections from reading and writing the document.

2. Cassandra:
   a. The "batch" operation guarantees atomic operations; however, there is a failure scenario in which this is not guaranteed.
   b. There is no batch isolation.
   c. Select statements cannot be included in a batch statement.

3. ReL:
   a. Uses Oracle RDF and SPARQL [*Oracle Graph* 2014)], which has full support for standard transaction properties, such as read consistency, serializable [Concurrency 2011], and ACID [Transactions 2011] transactions.
   b. Transactions can contain multiple insert, update, and select operations.
   c. With Oracle's multi-version, two-phase locking, writes never block reads.
   d. Locks occur at the row level, not at the database level.

In conclusion, using Oracle's Graph Database means the full features of transaction-level processing can be made available to NoSQL-like applications. As previously stated, this addresses a major weakness of current NoSQL database implementations.

There is a project at the University of Texas called the SALT project that is trying to address transaction capabilities for NoSQL database. The SALT project summarizes the situation with NoSQL databases 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**.

The BASE approach is in 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 lecture, "Shortcourse on Next Generation Systems" [Tolbert]. **It has also been shown in the paper that it is not necessary to abandon ACID transaction processing for NoSQL databases**.

4 SQL TO SPARQL DISCUSSION AND EXAMPLES

SQL and SPARQL are both powerful query languages; however, as shown in this paper, it is sometimes easier to write SQL instead of the equivalent SPARQL. To this end, ReL offers an SQL-to-SPARQL translation feature, which is invoked for each connection that is declared to be in "rdf_mode". For instance, the following SQL query in ReL:

```
SQL on conn "select deptno, sal from emp where SAL > 1000"
```
[14]

would be translated to the following SPARQL query:

```
SELECT v1 "DEPTNO", v2 "SAL"
 FROM TABLE(SEM_MATCH('SELECT ?v1 ?v2 WHERE {
?s1 rdf:type :EMP .
OPTIONAL { ?s1 :DEPTNO ?v1 }
OPTIONAL { ?s1 :SAL ?v2 }
```

[14] The standard Oracle scott/tiger schema is used in these examples [Scott Schema].

```
OPTIONAL { ?s1 :SAL ?v3 }
?s1 :SAL ?f1 .
FILTER(?f1 > 1000)
}
```

Here is a more complex example:

SQL:

```
SQL on conn "select dname, avg(sal) from emp e, dept d
where e.deptno = d.deptno
group by deptno
order by avg(sal) "
```

SPARQL:

```
SELECT v2 "DNAME", n1 "AVG(E.SAL)"
 FROM TABLE(SEM_MATCH('SELECT ?v2 (avg(?v3) as ?n1) WHERE {
       ?s1 rdf:type :EMP .
       ?s2 rdf:type :DEPT .
       OPTIONAL { ?s2 :DEPTNO ?v1 }
       OPTIONAL { ?s2 :DNAME ?v2 }
       OPTIONAL { ?s1 :SAL ?v3 }
       ?s1 :DEPTNO ?f1 .
       FILTER(?f1 = ?v1) }
GROUP BY  ?v2
ORDER BY ?v3' ,
SEM_MODELS('F2014_C##CS347_PROF'), null,
SEM_ALIASES( SEM_ALIAS('', 'http://www.example.org/people.owl#')), null) )
```

The algorithm for doing this translation is the subject of another paper.

Translating SQL to SPARQL is all well and good; however, there are more expressive query languages than SQL that can be translated to SPARQL. The SIM query language introduced by Doug Tolbert in his lecture, "Shortcourse on Next Generation Systems," at the Oregon Database Forum in February 1988 [Tolbert] is one such language. This language is being implemented in ReL with the hopes that it will be a more appropriate language for non-experts to use for things like data analysis. The SIM language is an object-oriented query language that supports the kind of database systems that will be discussed in Section 6, "INFERENCE DISCUSSION".

5 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 <u>any</u> environment that supports the CURL function. For instance, we use RESTful ReL in R [Project 2014] 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 emp table discussed in Section 4, "SQL TO SPARQL DISCUSSION AND EXAMPLES":

```
d <- getURL(URLencode('http://host:port/rest/native/?query="select * from emp"'),
httpheader=c(DB='jdbc:oracle:thin:@host:port:sid', USER='user', PASS='password',
MODE='rdf_mode', MODEL='model', returnFor = 'R'), verbose = TRUE)
```

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

```
print(d)

[1] "list(EMPNO=c(9999, 7369, 7499, 7521, 7566, 7654, 7698, 7782, 7788, 7839, 7844, 7876,
7900, 7902, 7934),ENAME=c('PHIL', 'SMITH', 'ALLEN', 'WARD', 'JONES', 'MARTIN', 'BLAKE',
'CLARK', 'SCOTT', 'KING', 'TURNER', 'ADAMS', 'JAMES', 'FORD', 'MILLER'),JOB=c('CLERK',
'CLERK', 'SALESMAN', 'SALESMAN', 'MANAGER', 'SALESMAN', 'MANAGER', 'MANAGER', 'ANALYST',
'PRESIDENT', 'SALESMAN', 'CLERK', 'CLERK', 'ANALYST', 'CLERK'),MGR=c('null', 7902, 7698,
7698, 7839, 7698, 7839, 7839, 7566, 'null', 7698, 7788, 7698, 7566, 7782),HIREDATE=c('null',
'1980-12-17 00:00:00', '1981-02-20 00:00:00', '1981-02-22 00:00:00', '1981-04-02 00:00:00',
```

```
'1981-09-28 00:00:00', '1981-05-01 00:00:00', '1981-06-09 00:00:00', '1982-12-09 00:00:00',
'1981-11-17 00:00:00', '1981-09-08 00:00:00', '1983-01-12 00:00:00', '1981-12-03 00:00:00',
'1981-12-03 00:00:00', '1982-01-23 00:00:00'),SAL=c('null', 800.0, 1600.0, 1250.0, 2975.0,
1250.0, 2850.0, 2450.0, 3000.0, 5000.0, 1500.0, 1100.0, 950.0, 3000.0, 1300.0),COMM=c('null',
'null', 300.0, 500.0, 'null', 1400.0, 'null', 'null', 'null', 'null', 'null', 'null', 'null',
'null', 'null'),DEPTNO=c('null', 20, 30, 30, 20, 30, 30, 10, 20, 10, 30, 20, 30, 20, 10))"
```

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, 2^31-1))))
```

Finally, head(df) results in the following:

```
head(df)

  EMPNO  ENAME      JOB  MGR            HIREDATE  SAL COMM DEPTNO
1  9999   PHIL    CLERK null                null null null   null
2  7369  SMITH    CLERK 7902 1980-12-17 00:00:00  800 null     20
3  7499  ALLEN SALESMAN 7698 1981-02-20 00:00:00 1600  300     30
4  7521   WARD SALESMAN 7698 1981-02-22 00:00:00 1250  500     30
5  7566  JONES  MANAGER 7839 1981-04-02 00:00:00 2975 null     20
6  7654 MARTIN SALESMAN 7698 1981-09-28 00:00:00 1250 1400     30
```

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].[15]

[15] Dr. Cannata and the students in his Data Science classes at the University of Texas at Austin are implementing a DBaaS Cloud environment on Oracle's Public Cloud using the RESTful ReL technology discussed in this paper. A large set of publically available data will be hosted in this environment for visualization and data mining analysis.

6 INFERENCE DISCUSSION

In his "Shortcourse on Next Generation Systems" [Tolbert], Doug Tolbert uses the following database schema for a set of examples:

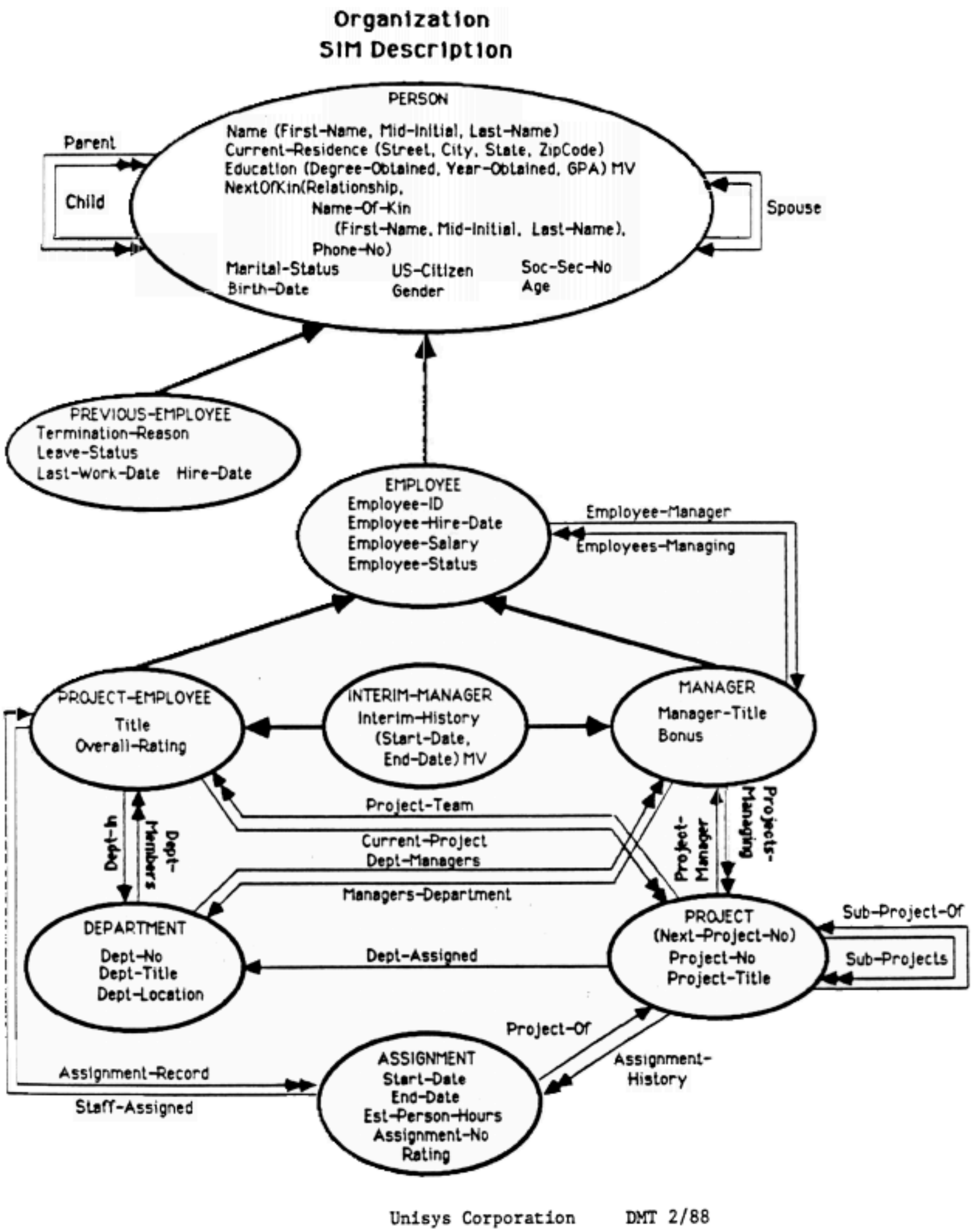

Figure 9

This schema shows a simple organizational structure in which it is important to notice that all of the relationships are bi-directional (e.g., the parent-child relationship on the Person class) in a manner discussed in the Introduction using the "InverseOf" inference. The entities are represented in a class hierarchy with inheritance. This type of schema served as the data model for the SIM data management system that was built by Boroughs Corporation in the 1970s. There was no transforming this schema to a Relational data model before implementing the database, i.e., inference would help remove the need to convert a conceptual model to a Relational model when building a database application, which is a widespread practice in the industry.[16] Because of this, Tolbert argued that more of the semantics of the data were incorporated into the database, which we believe is a highly desirable goal and what should be the motivation for new data management systems.

The examples Tolbert worked through in his paper showed a comparison of SQL queries and SIM queries for the Organization schema as implemented in a Relational database and directly in the SIM database. One of the more complex queries he showed was:

"Print the names of employees and the titles of all their projects if they work on any project assigned to the Accounting Department."

Tolbert showed the following SQL query:

```
SELECT First-Name, Mid-Initial, Last-Name, Project-Title
  FROM Person, Project-Person, Project
 WHERE Person.Soc-Sec-No = Project-Person.Soc-Sec-No
   AND Project.Project-No = Project-Person.Project-No
   AND EXISTS
     ( SELECT *
         FROM Project-Person, Project, Department
        WHERE Project-Person.Soc-Sec-No = Person.Soc-Sec-No
          AND Project-Person.Project-No =
                  Project.Project-No
          AND Department.Dept-No = Project.Dept-No
          AND Department.Dept-Title = "Accounting")
```

Tolbert also showed the following SIM query:

```
RETRIEVE Name of Project-Employee,
         Project-Title of Current-Project
WHERE Dept-Title of SOME (Dept-Assigned of Current-Project)
      = "Accounting"
```

Here, "Name of Project-Employee" is inherited from the "Person" class, "Project-Title" is retrieved by traversing the "Current-Project" relationship using the "of" operation, and "Dept-Title" is retrieved by traversing the "Current-Project" relationship and then the "Dept-Assigned" relationship.

This example query and the others in the paper show the benefit of inheritance and bi-directional relationships being directly implemented in the data management system as the application logic becomes much simpler. NoSQL databases are not moving in the direction of having these capabilities. Inference in RDF and OWL can be used to prototype these capabilities today. In addition, there are many more inference capabilities that can be used to prototype a new data management system capable of the following inferences.

Functional Property:

1. Fact: Bob has manager Mary.
2. Fact: Bob has manager Maria.
3. Inference: Mary is same as Maria.
4. Inference: Maria is same as Mary.

---

16 When building a database application, Dr. Cannata teaches his Data Management students to first build a Conceptual Model that reflects the customer's requirements (this is similar to Tolbert's Organization schema shown in Figure 9). This Conceptual Model usually contains Class Hierarchies because, going back to the time of Aristotle, that is the best way to model the real world. Next, Dr. Cannata has his students to convert the Conceptual Model to a Logical Model in which each class hierarch is rolled up into one entity class (in Tolbert's schema the EMPLOYEE, PROJECT-EMPLOYEE, INTERIM-MANAGER, and MANAGER classes would be rolled up into the PERSON class). Then, Dr. Cannata has his students convert the Logical Model to a Relational Model from which the database DDL statements are generated. Dr. Cannata also has his students build views and update triggers for each of the classes that were rolled up into the top-level class. All of this work would be unnecessary if the database management system has Owl-like inference and understands the Conceptual Model.

The Functional Property could be used for integrating data from two different databases: one where Mary is said to be Bob's manager; the other where Maria is said to be Bob's manager, when, in fact, Bob has just one manager who is referred to by both names.

Transitive Property:

1. Fact: Bob has ancestor Mary.
2. Fact: Mary has ancestor Tom.
3. Inference: Bob has ancestor Tom.

Many of these capabilities are being prototyped in ReL along with the SIM query language on top of SPARQL, but that is the subject of another paper.

7 SUMMARY

In an InfoWorld article [Oliver 2014], the author claims, "the time for NoSQL standards is now." But this paper has shown that the already standardized RDF/OWL and SPARQL based data management systems are an attractive option where current NoSQL implementations have failed to fill the bill in certain domains due to their lack of join support, lack of full ACID transaction support, and a missing standardized query language.

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.

This paper also touched on the potential of inference as part of the data management system. With notions for defining object types and consistent bi-directional relationships via "of type" and "inverse of" like semantics. Having such relationships and knowledge in the data allows for easier application development by reducing the complexity of the application trying to store and retrieve data.This knowledge representation also allows the conceptual data model to become the database management system's schema, whether explicit or implicit, reducing the requirement to convert the conceptual models to relational models.

8 REFERENCES

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

[2] *Apache Cassandra*. (2014). Retrieved from http://cassandra.apache.org/

[3] 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

[4] Cannata, P. (1991). The Irresistible Move Towards Interoperable Database Systems. *Keynote Address at the First International Workshop on Interoperability in Multidatabase Systems.* Kyoto, Japan.

[5] *Carnot Home Page*. (n.d.). Retrieved from http://web.inet-tr.org.tr/History/einet/MCC/Carnot/carnot.paper.html

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

[7] *Cassandra Atomic*. (n.d.). Retrieved from http://www.datastax.com/dev/blog/atomic-batches-in-cassandra-1-2

[8] *Cassandra Batch*. (n.d.). Retrieved from http://www.datastax.com/documentation/cql/3.0/cql/cql_reference/batch_r.html

[9] *Cassandra Isolation*. (n.d.). Retrieved from http://www.datastax.com/documentation/cql/3.0/cql/cql_reference/batch_r.html

[10] 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

[11] *CloudCredo*. (n.d.). Retrieved from http://www.cloudcredo.com/cloudcredo-dbaas-demo/

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

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

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

[15] *Flask*. (2014). Retrieved from Flask web development, one drop at a time: http://flask.pocoo.org/

[16] *Flask-RESTful*. (2014). Retrieved from http://flask-restful.readthedocs.org/en/0.3.0/

[17] *Hadoop*. (n.d.). Retrieved from http://hadoop.apache.org/

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

[19] Loshin, D. (n.d.). *An Introductin to NoSQL Data Management for Big Data*. Retrieved from http://data-informed.com/introduction-nosql-data-management-big-data/

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

[21] *Mongo lock granularity*. (n.d.). Retrieved from http://docs.mongodb.org/manual/faq/concurrency/

[22] *MongoDB*. (2014). Retrieved from MongoDB: http://www.mongodb.org

[23] *MongoDB ACID*. (n.d.). Retrieved from http://css.dzone.com/articles/how-acid-mongodb

[24] *MongoDB Aggregation*. (2014). Retrieved from http://docs.mongodb.org/manual/core/aggregation-introduction/

[25] *MongoDB Transactions*. (n.d.). Retrieved from http://www.tokutek.com/2013/04/mongodb-transactions-yes/

[26] *MongoDBRefs*. (n.d.). Retrieved from http://docs.mongodb.org/manual/reference/database-references/

[27] *Mongolab*. (2014). Retrieved from https://mongolab.com/

[28] Munindar Singh, C. T. (1994). *SIGMOD '94 Proceedings of The 1994 ACM SIGMOD international conference on Management of data.*

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

[30] 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

[31] *Oracle Graph*. (2014). Retrieved from http://www.oracle.com/us/products/database/options/spatial/overview/index.html

[32] *Oracle Public Cloud*. (n.d.). Retrieved from https://cloud.oracle.com/home

[33] OWL. (2014). Retrieved from http://www.w3.org/TR/owl2-primer/

[34] *OWL Inference*. (n.d.). Retrieved from http://www.w3.org/standards/semanticweb/inference

[35] Paul Attie, M. S. (1993). *Proceedings of teh 19th VLDB, Dublin, Ireland.*

[36] Project, R. (2014). Retrieved from http://www.r-project.org/

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

[38] *Scott Schema*. (n.d.). Retrieved from http://www.cs.utexas.edu/~cannata/dbms/Class%20Notes/02%20emp_dept.ddl

[39] 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

[40] SPARQL. (2014). Retrieved from http://www.w3.org/TR/sparql11-overview/

[41] *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

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

[43] Tilkov, S. (n.d.). *A Brief Introduction to REST*. Retrieved from http://www.infoq.com/articles/rest-introduction

[44] Tolbert, D. (n.d.). Retrieved from http://www.cs.utexas.edu/~cannata/dbms/Class%20Notes/09%20Shortcourse%20on%20Next%20Generation%20Database%20Systems.pdf

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

Appendix A: CREATING THE DATABASES

MongoDB

For Mac, see http://docs.mongodb.org/manual/tutorial/install-mongodb-on-os-x/
For Windows, see http://docs.mongodb.org/manual/tutorial/install-mongodb-on-windows/

Here's a summary of what worked on the Mac:

Install MongoDB:

```
MacBook-Pro:~ $ brew update
MacBook-Pro:~ $ brew install mongodb
MacBook-Pro:~ $ mkdir –p /data/db
```

Start MongoDB daemon:

```
MacBook-Pro:~ $ sudo mongod
```

In another terminal window, start mongo:

```
MacBook-Pro:~ $ mongo
MongoDB shell version: 2.6.6
connecting to: test
> use books
switched to db books
> db.createCollection("mybooks")
{ "ok" : 1 }
```

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

To run the application:

pip install --target="/Users/pcannata/Mine/MyReL/Papers/MongoDB Paper/bookdb-master/BookFlask2" flask
pip install --target="/Users/pcannata/Mine/MyReL/Papers/MongoDB Paper/bookdb-master/BookFlask2" pymongo
python books.py

Making a Cassandra Connection and Create the Triple-Store Table If It Doesn't Already Exist

Import Cluster from Cassandra.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

```
conn = connectTo 'jdbc:oracle:thin:@host:1521:orcl' 'user' 'password' 'rdf_mode'
'rdf_model'
```
[17]

[17] Creating a connection also creates an RDF Model and some utility sequences if the model does not already exist 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")
```

```
    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[18] (data-level triples are bold and OWL-level triples are italicized and underlined):

```
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'));
```

[18] URIs have been abbreviated to help with readability.
[19] 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#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 `New ReL search code below:`

```
@app.route('/search/', methods=['GET', 'POST'])
def search():
```

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

        no_results = result_dict == 0
        return render_template('search.html', posting=True, query=query,
no_results=no_results, results=result_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 display(title, author):
	if request.method == 'GET':
		cursor = books.find_one({'title':title, 'author':author})
	elif request.method == 'POST':
		cursor = update(title, author)
	results = {field: value for field, value in cursor.items()}
	js_results = {str(field).replace('"', '\\"') :str(value).replace('"',
	                         '\\"') for field, value in results.items()}
	if request.method == 'POST' and (title != request.form['title'] or
	                                  author != request.form['author']):
		return redirect('/detail/'+request.form['title']
		                                  +'/'+request.form['author']+'/')
	else:
		return render_template('detail.html', result=results,
		                                            js_results=js_results)


#Update a book's fields and attributes
def update(title, author):
	#Add new values of all pre-existing attributes
	updated_document = {attribute: value for attribute, value in
	        request.form.iteritems() if attribute[:14] != '__new__field__'
	        and attribute[:14] != '__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

	return books.find_and_modify({'title':title, 'author': author},
	                                       updated_document, new=True)
```

Cassandra Code

```
#Update information for a book
@app.route('/detail/<id>/', methods=['GET', 'POST'])
def display(id):

	id = uuid.UUID(id)
	if request.method == 'POST':
		update(id)

	results = {}
	all_props_and_vals = session.execute("SELECT property, value FROM
	                              "+table_name+" WHERE id = %s", (id,))
	for property in all_props_and_vals:
		results[property.property] = property.value
	js_results = {str(field).replace('"', '\\"') :str(value).replace('"',
	                         '\\"') for field, value in results.items()}
	return render_template('detail_cass.html', result=results,
	                                      js_results=js_results, id=id)


def update(id):
```

```
        old_prop_query = "SELECT property FROM "+table_name+" WHERE id=%s"
        old_rows = session.execute(old_prop_query, (id,))
        #all properties for this book prior to upgrade
        old_properties = {str(row.property) for row in old_rows}
        #all properties for this book after upgrade
        current_properties = set()

        # 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.

        batch = BatchStatement()
        for key, value in request.form.iteritems():
              #add new property and value to book
              if key[:14] == '__new__field__':
                    pair_number = key[14:]
                    batch.add("UPDATE "+table_name+" SET value = %s WHERE id =
                                                        %s and property =
                       %s",(request.form['__new__value__'+str(pair_number)],
                                                              id, value))
                    current_properties.add(str(value))

              #update value of existing property of book
              elif key[:14] != '__new__value__':
                    batch.add("UPDATE "+table_name+" SET value = %s WHERE id =
                                      %s and property = %s",(value, id, key))
                    current_properties.add(str(key))

        to_remove = old_properties - current_properties
        delete_statement = "DELETE FROM "+table_name+" WHERE id=%s and
                                                             property=%s"
        for property in to_remove:
              batch.add(delete_statement, (id, property))
        session.execute(batch)
```

ReL Code

```
@app.route('/detail/<title>/<author>/', methods=['GET', 'POST'])
def display(title, author):
      if request.method == 'GET':
            results = SQL on conn """select * from books where title =
                             '"""title"""' and author = '"""author"""'"""
            result_dict = {}
            num = 0
            for r in results[1] :
               if results[0][num] != 'DBUNIQUEID' :
                  result_dict.update({results[0][num] : r})
               num += 1
      elif request.method == 'POST':
          return update(title, author)
      return render_template('detail.html', result=result_dict,
                                                   js_results=result_dict)

#Update a book's fields and attributes
def update(title, author):
      subject = SQL on conn2 """SELECT s1
                   FROM TABLE(SEM_MATCH('SELECT * WHERE {
                         ?s1 rdf:type :books .
                         OPTIONAL { ?s1 :title ?v1 }
                         OPTIONAL { ?s1 :author ?v2 }
                         ?s1 :title ?f1 .
                         ?s1 :author ?f2 .
```

```
                              FILTER(?f1 = \""""title"""\" && ?f2 =
                                                          \""""author"""\") }' ,
                    SEM_MODELS('BOOK_C##CS347_PROF'), null,
                    SEM_ALIASES( SEM_ALIAS('',
                        'http://www.example.org/people.owl#')), null) ) """

# Add new values of all pre-existing attributes
      results = SQL on conn """select * from books where title =
                                 '"""title"""' and author = '"""author"""'"""
      result_dict = {}
      num = 0
      for r in results[1] :
            if results[0][num] != 'DBUNIQUEID' :
               result_dict.update({results[0][num] : r})
            num += 1
      updated_dict = {attribute: value for attribute, value in
                                request.form.iteritems() if attribute[:14] !=
                    '__new__field__' and attribute[:14] != '__new__value__'}
      changes_dict = dict([(k, updated_dict.get(k)) for k in updated_dict if
                            updated_dict.get(k) not in result_dict.values()])
      removes_dict = dict([(k, result_dict.get(k)) for k in result_dict if
                            result_dict.get(k) not in updated_dict.values()])
      for k in changes_dict :
         do_update[20](subject, k, result_dict.get(k), changes_dict.get(k))

# Add values of new fields, if any
      num_old_fields = len(updated_dict)
      num_new_fields = (len(request.form)-num_old_fields)/2
      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_dict[new_attribute] = new_value
                  print subject[1][0], new_attribute, new_value
                  do_insert(subject, new_attribute, new_value)

# Remove selected attributes, if any
      for k in removes_dict :
         do_remove(subject, k, removes_dict.get(k))

# Return new results
      results = SQL on conn """select * from books where title =
                                 '"""title"""' and author = '"""author"""'"""
      result_dict = {}
      num = 0
      for r in results[1] :
         if results[0][num] != 'DBUNIQUEID' :
            result_dict.update({results[0][num] : r})
         num += 1
      return render_template('detail.html', result=result_dict,
                                                       js_results=result_dict)
```

[20] The code for the "do_update", "do_insert", and "do_remove" functions can be found in the GitHub repository. The functionality of these functions is being incorporated into the ReL SQL language.