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 (Samsung Austin Research Center and the University of Texas), 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 a promising alternative to existing 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.

Table of Contents:

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] 2014 Forrester Research, Inc.

1. A 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 their perceived need for a special kind of "scaleout"[4] (i.e., so called "inexpensive" scaleout).
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 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.

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:

---

[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 Discussion" 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)

[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.

1. SPARQL is a full featured, international standard query language. It supports all of the features of an ad hoc query language such as projection of attributes, selection of instances, joins, aggregation, and sub-queries. This paper will also demonstrate 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. This would seem ideal as a query language for NoSQL databases.
2. RDF/OWL and SPARQL systems support "joins" in the database, no application processing is required for joins (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 scaleup and scaleout 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.

1. 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)
2. Support for 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)[6]
3. 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 will be demonstrated in a case study presented in this paper 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 will be discussed in Section 4.

Lastly, this paper touches on the game-changing potential of Inference (OWL Inference) integration into data management systems, which is an existing feature in RDF/OWL based systems. Inference is means of deriving logical conclusions from facts already known in the system. The example below helps show 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 for more details).

[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)

Another example follows:

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.

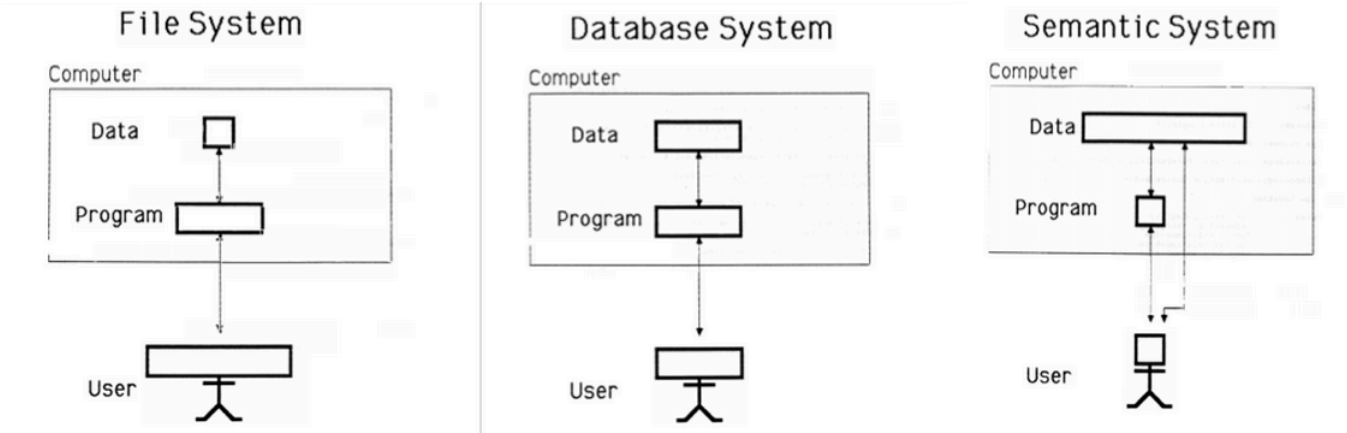
This is an important relationship for the support of class inheritance (see Section 6 for more details).

As mentioned earlier, Inference is natively supported in OWL-based systems, but 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 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).

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** (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). 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, leaving the application to deal solely with application level functionality.

Today, with Relational systems, we're barely in the middle; with NoSQL databases and Hadoop[7]-like clusters, we're moving to the left. **We need to be moving to the right**!



[7] "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.

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 better than Relational technology.

2. Case Study

A case study looking at the implementation of a Flask-based library website with simple CRUD[8] 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[9]. A 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 implementation can be found at https://github.com/IsabellaBhardwaj/bookdb/tree/master/.

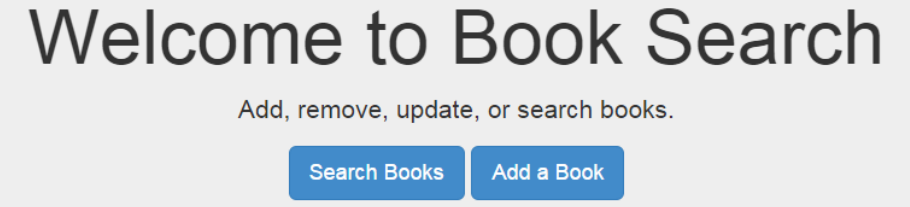


Figure 1

2.1 Inserting book data into the library website database

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

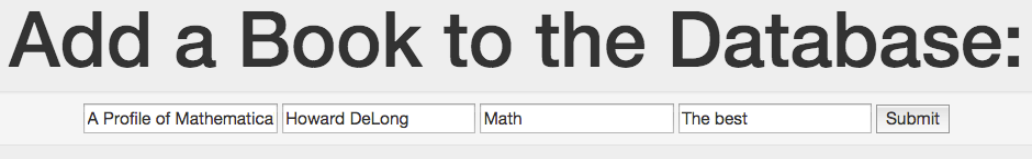


Figure 2

[8] Create, Retrieve, Update, Delete.

[9] 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.

The code for adding a book for each implementation is shown in Appendix C. A high level summary of the code follows:

A. 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)

Also, "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)

B. For the Cassandra application

In the Cassandra application, we use the Cassandra database as a triple-store[10]. 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 in a "batch" statement (Cassandra Batch) 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)
```

[10] 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.

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). 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.” (Cassandra Isolation)

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

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
```

In ReL, the full SQL statement is a concatenation of the python string beginning with “insert” and the interpreters evaluation of 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 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 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

The code for searching the database for each implementation is 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 where "query" is a string containing the terms the user typed into the search bar:

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

B. For the Cassandra applicaiton

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
```

C. For the ReL application

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

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

titles[11] = 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
```

Behind the scenes, ReL converts the SQL select statements into SPARQL statements, as shown below[12,13]:

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

```
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

To update the information about an existing book in the database, the user first enters the title or author into the search bar. Figures 4.1 – 4.4 show the sequence of menus that follow.

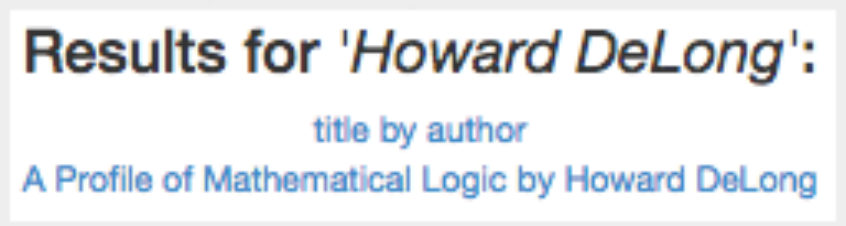


Figure 4.1 – Next, click on the link for the book.

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

[13] 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)

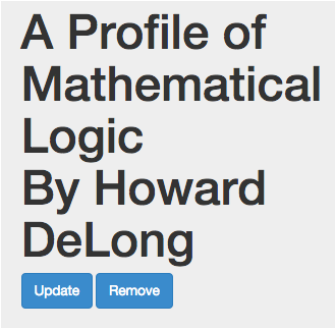


Figure 4.2 – Next, click on Update.

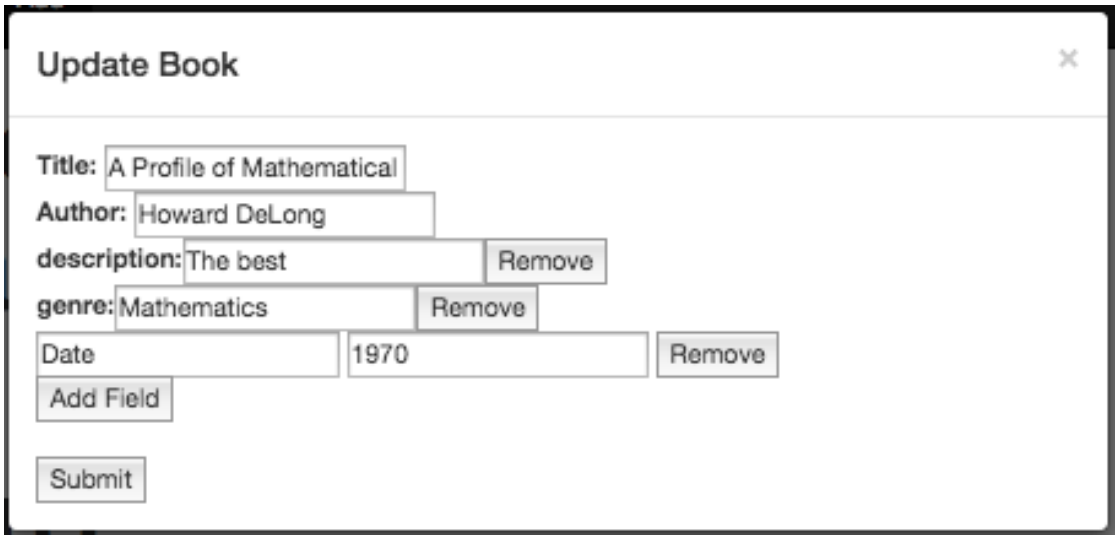


Figure 4.3 – Next, edit the information. I this case Math was changed to Mathematics, and a Date field was added.

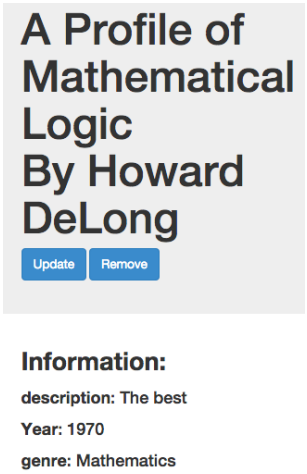


Figure 4.4 – After clicking on the Submit button, the updated information is displayed.

The code for updating the database for each implementation is 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 "update" 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:

```
books.update({'title':title, 'author': author}, updated_document)
```

Then, a query is performed to display the updated results.

```
books.find_one({'title': request.form['title'], 'author':
                                                    request.form['author']})
```

Since MongoDB provides transactions for only a single statement, these two operations cannot be combined into a single transaction. This means that updates from other sessions not just the update from this session could appear in the final display of the data. This might be desirable in this application but it is not generally desirable.

There's actually an operation called findAndModify (http://docs.mongodb.org/manual/reference/command/findAndModify/) that updates a document atomically and then returns that document (but you can't specify the write concern so it's not guaranteed to be durable). By using findAndModify, the mongodb app will display only the changes the current user made to a book, without displaying any changes made at the same time by any other user. Dr. Cannata, I have updated this function in the branch return_document to use findAndModify, and if you want to use this version in the paper I'll merge it with master. findAndModify works okay for the book app's purposes because only one document is modified, but it can be pointed out that findAndModify would not work in many cases because it can only update and return one document, and you can't specify the write concern of the operation when using it.

B. For the Cassandra application

In the Cassandra application, the following statement that will perform the addition of the "Year" field is prepared for "batch" mode[14] execution:

```
batch.add("UPDATE "+table_name+" SET value = %s WHERE id = %s and property =
          %s",(request.form['__new__value__'+str(pair_number)], id, value))
```

Then, the following statement that will perform the update, changing the value of" genre" field from "Math" to "Mathematics" is prepared for "batch" mode execution:

[14] "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)

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

Then the "batch" is executed:

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

After this, the following query returns the updated information to the web application:

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

Notice, 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.

C. 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.o
wl>', '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. Transaction 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 stated earlier in the paper, this addresses a major weakness of current NoSQL database implementations which is also seen in this example.

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 "Shortcourse on Next Generation Systems" (Tolbert) and **it has 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 has been show in this papers, it's sometime easier and more compact 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"
```
[15]

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

A more complete example would be:

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 .
```

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

```
        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 "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.

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 any 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 the previous section:

```
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)
```

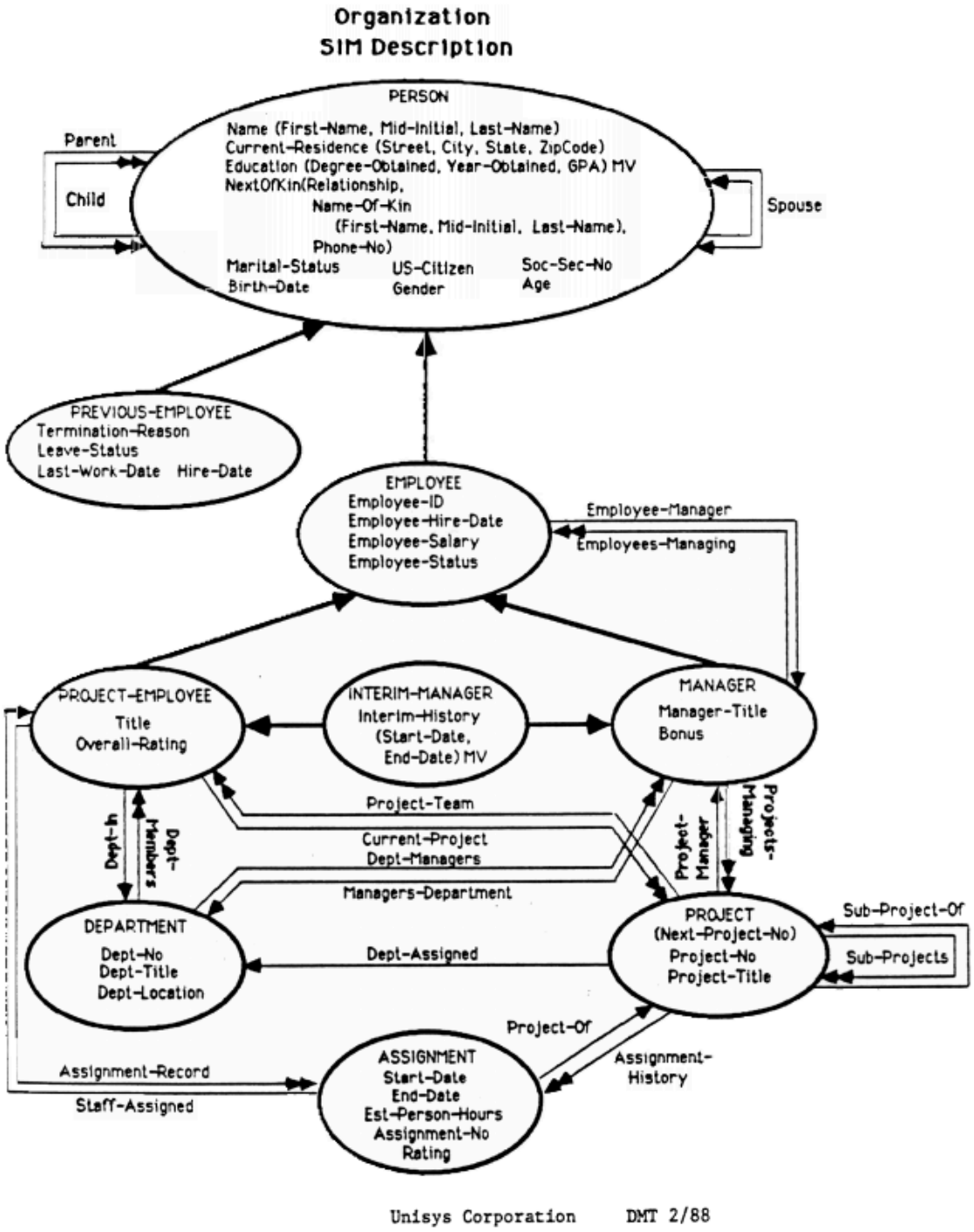
```
  EMPNO  ENAME      JOB  MGR            HIREDATE  SAL COMM DEPTNO
1  9999   PHIL    CLERK null                null null null   null
2  7369  SMITH    CLERK 7902 1980-12-17 00:00:00  800 null     20
3  7499  ALLEN SALESMAN 7698 1981-02-20 00:00:00 1600  300     30
4  7521   WARD SALESMAN 7698 1981-02-22 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)[16].

[16] 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 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:



This schema shows a simple organization 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 earlier using the "InverseOf" inference and 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 70s. There was no transforming this schema to a Relational data model before implementing the database. Because of this, Doug 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 Doug 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."

The SQL Doug showed for this query was:

```
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")
```

The SIM Doug showed for this query was:

```
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 and the others in the paper make clear the fundamental importance of inheritance and bi-directional relationships being directly implemented in the data management system. This, at a minimum, is what modern data management systems should provide but they don't. NoSQL databases are not moving in the direction of having these capabilities. Inference in RDF and OWL can be used to at least prototype these easily capabilities. In addition, there are many more Inference capabilities that can also be used to at prototype new data management capabilities such as:

Functional Property:

1. Fact: Bob has mother Mary

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

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 this 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. 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.

This paper also highlighted that 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 database management system's schema whether explicit or implicit.

8. Bibliography

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

Apache Cassandra. (2014). Retrieved from http://cassandra.apache.org/

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

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

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

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

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

Cassandra Batch. (n.d.). Retrieved from http://www.datastax.com/documentation/cql/3.0/cql/cql_reference/batch_r.html
Cassandra Isolation. (n.d.). Retrieved from http://www.datastax.com/documentation/cql/3.0/cql/cql_reference/batch_r.html
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
CloudCredo. (n.d.). Retrieved from http://www.cloudcredo.com/cloudcredo-dbaas-demo/
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/
Flask-RESTful. (2014). Retrieved from http://flask-restful.readthedocs.org/en/0.3.0/
Hadoop. (n.d.). Retrieved from http://hadoop.apache.org/
Hammer, M., & McLeod, D. (1981). Database Description with SDM: A Semantic Database Model. *ACM Transaction of Database , 6* (3).
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/
McDonald, C., Blevins, L., Tierney, W., & Martin, D. (1988). The Regenstrief Medical Records. *MD Computing (5(5))* , 34-47.
Mongo lock granularity. (n.d.). Retrieved from http://docs.mongodb.org/manual/faq/concurrency/
MongoDB. (2014). Retrieved from MongoDB: http://www.mongodb.org
MongoDB ACID. (n.d.). Retrieved from http://css.dzone.com/articles/how-acid-mongodb
MongoDB Aggregation. (2014). Retrieved from http://docs.mongodb.org/manual/core/aggregation-introduction/
MongoDB Transactions. (n.d.). Retrieved from http://www.tokutek.com/2013/04/mongodb-transactions-yes/
MongoDBRefs. (n.d.). Retrieved from http://docs.mongodb.org/manual/reference/database-references/
Mongolab. (2014). Retrieved from https://mongolab.com/
Munindar Singh, C. T. (1994). *SIGMOD '94 Proceedings of The 1994 ACM SIGMOD international conference on Management of data.*
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
Oracle Public Cloud. (n.d.). Retrieved from https://cloud.oracle.com/home
OWL. (2014). Retrieved from http://www.w3.org/TR/owl2-primer/
OWL Inference. (n.d.). Retrieved from http://www.w3.org/standards/semanticweb/inference
Paul Attie, M. S. (1993). *Proceedings of teh 19th VLDB, Dublin, Ireland.*
Project, R. (2014). Retrieved from http://www.r-project.org/
RDF. (2014). Retrieved from http://www.w3.org/TR/2014/NOTE-rdf11-primer-20140225/

Scott Schema. (n.d.). Retrieved from http://www.cs.utexas.edu/~cannata/dbms/Class%20Notes/02%20emp_dept.ddl
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.
Tilkov, S. (n.d.). *A Brief Introduction to REST*. Retrieved from http://www.infoq.com/articles/rest-introduction
Tolbert, D. (n.d.). Retrieved from http://www.cs.utexas.edu/~cannata/dbms/Class%20Notes/09%20Shortcourse%20on%20Next%20Generation%20Database%20Systems.pdf
Transactions. (2011). Retrieved from http://docs.oracle.com/cd/E28271_01/server.1111/e25789/transact.htm#g11401

Appendix A: Creating the Databases

MongoDB

For the 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 Database

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

```
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

```
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 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")
    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 shown in bold below and OWL level triples are shown in italics and underlined).

```
BEGIN
commit ;
set transaction isolation level serializable[19] ;
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'));
```

[18] URIs have been abbreviated to help with readability.

[19] 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', '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

```
#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 :
```

[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.

```
        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)
```