Utilizing Nested Normal Form to Design Redundancy Free JSON Schemas

https://doi.org/10.3991/ijes.v4i4.6539

Wai Yin Mok
University of Alabama in Huntsville, Huntsville, AL, USA

***Abstract*—JSON (JavaScript Object Notation) is a lightweight data-interchange format for the Internet. JSON is built on two structures: (1) a collection of name/value pairs and (2) an ordered list of values (http://www.json.org/). Because of this simple approach, JSON is easy to use and it has the potential to be the data interchange format of choice for the Internet. Similar to XML, JSON schemas allow nested structures to model hierarchical data. As data interchange over the Internet increases exponentially due to cloud computing or otherwise, redundancy free JSON data are an attractive form of communication because they improve the quality of data communication through eliminating update anomaly. Nested Normal Form, a normal form for hierarchical data, is a precise characterization of redundancy. A nested table, or a hierarchical schema, is in Nested Normal Form if and only if it is free of redundancy caused by multivalued and functional dependencies. Using Nested Normal Form as a guide, this paper introduces a JSON schema design methodology that begins with UML use case diagrams, communication diagrams and class diagrams that model a system under study. Based on the use cases' execution frequencies and the data passed between involved parties in the communication diagrams, the proposed methodology selects classes from the class diagrams to be the roots of JSON scheme trees and repeatedly adds classes from the class diagram to the scheme trees as long as the schemas satisfy Nested Normal Form. This process continues until all of the classes in the class diagram have been added to some JSON scheme trees.**

***Index Terms*—JSON, Nested Normal Form, Redundancy Free JSON Scehems.**

I. INTRODUCTION

JSON (JavaScript Object Notation), based on the JavaScript programming language, is a lightweight data-interchange format for the Internet. JSON is built on two structures: (1) a collection of name/value pairs and (2) an ordered list of values (http://www.json.org/). Virtually all modern programming languages support these simple data structures in one form or another. Because of this simple and straightforward approach, JSON is easy to use and it has the potential to become the data-interchange format of choice for the Internet. Maintained by The Object Management Group (http://www.omg.org/), Unified Modeling Language (http://www.uml.org/) is the standard modeling language of the software industry. UML not only provides a number of diagrams to specify, visualize, and document software systems, UML is also a methodology that analysts are guided by its design principles when constructing UML diagrams. Of all the UML diagrams, use case diagrams, communication diagrams, and class diagrams are particularly relevant to this research. Use case diagrams specify the main functions of a system under study. Through use case diagrams, communication diagrams can be derived, which document the data that are passed between the users and the system when a use case is carried out. As a result of use case diagrams and communication diagrams, data access patterns can be discovered.

Given one or more UML class diagrams, use case diagrams and communication diagrams for a system under study, this paper presents an algorithm that generates JSON schemas from these diagrams that are free from redundancy. This approach has several advantages: (1) UML models are designed to capture the relevant aspects of a system under study. In addition, UML is also a design methodology such that the diagrams are constructed according to sound design principles. (2) Using UML diagrams as the starting point facilitate automatic JSON schema generation because there are numerous UML design software tools and the algorithm presented in this paper can be easily integrated into these design tools.

Like XML schema design research, related work on JSON schema has started to appear (http://json-schema.org/). This research is different from those described in http://json-schema.org/ because this research begins with UML diagrams but those described in http://json-schema.org/ mostly generate JSON schema in a reverse-engineering manner when JSON data are given.

The rest of the paper is organized as follows. Section II presents a motivating example and Section III shows the foundational definitions. The JSON schema generation algorithm is shown in Section IV and we conclude in Section V.

II. A MOTIVATING EXAMPLE

Figure 1 shows a use case diagram. It has five use cases, which represents the five main functions provided by the course management system under study. It also has four users, one of which is non-human. Figure 1 also shows the use cases that are used by each user. For example, a student is only concerned with the use case "register for courses," but a lecturer is associated with the use cases "select courses to teach" and "request course roster." A registrar, on the other hand, will have access to three different use cases and the sole non-human user, the billing system, needs access to only one use case.

Figure 2a shows a communication diagram that documents the data that are passed between a student and the course management system when the use case "register for courses" is carried out. The communication diagram also documents the order of the data that are passed between them. The student first enters his/her name and student ID to the CMS. Then, the CMS gives permission to the student to login. After that, the student enters the course ID of the course for which he/she wants to register. The CMS then passes on the course ID to the database to check if there is any available seat. If so, the CMS will allow the student to register. Figure 2b shows another communication diagram, which documents the data passed between a lecturer and the CMS when the use case "request course roster'' is executed. First, the lecturer enters his/her name and ID. The CMS then allows the lecturer to login. After that, the lecturer enters the ID of the course whose roster the lecturer requests and finally the roster is returned to the lecturer.

Given a use case diagram that documents the main functions of a system and the communication diagrams that specify the data passed between the different entities when the system's use cases are carried out, data access patterns can be discovered. For example, Figure 2a dictates that the name and ID of a student is first accessed, and then a course ID and the course's information. Finally, the registered courses are stored along with the student to facilitate future searches on the courses taken by the student. Figure 2b dictates that the name and ID of a lecturer are first accessed, and then a course ID, and finally the roster of the course. To facilitate execution of these use cases, the data used in a use case should be clustered together in a JSON schema to reduce query time and data transfer time. For example, a JSON schema may cluster the student information for each student to facilitate searches on name and student ID. After registering for courses, the registered courses should also be clustered along with each student in a JSON schema to facilitate retrieval of courses when the student ID is given. On the other hand, to speed up the execution of the use case "request course roster," the roster of each course should be stored along with the course in a JSON schema. Hence, different use cases may lead to different JSON schema designs, and a balance must be maintained among competing designs. We believe the best solution is based on the use cases' execution frequencies. For example, if the use case "register for courses" will be executed much more frequently than the use case "request course roster," then we should design the JSON schemas that favor the use case "register for courses."

Class diagrams, which describe the structure of a system by showing the system's classes, their attributes, operations (or methods), and the relationships among objects, are also relevant to this research. Fig. 3 shows a class diagram, in which there are a class Student, a class Course, and a class Lecturer. The attributes of each class are also shown in the figure. The main purpose of class diagrams is to show the static aspects of the system. In this regard, class diagrams are different from communication diagrams because communication diagrams show the system's dynamic aspects.

Like XML (EXtensible Markup Language), which was designed to store and transport data, JSON is hierarchical because nested data are allowed. We now introduce JSON scheme trees, whose formal definition will be presented later. JSON scheme trees are JSON schemas in tree form

to show the nesting of the data. Two sets of redundancy-free JSON scheme trees are shown in Figures 4a and 4b for the class diagram in Figure 3. Which set is the best for the system depends on the prevalent use case of the system. As an example, if "register for courses" is the prevalent use case, then the scheme trees in Figure 4a

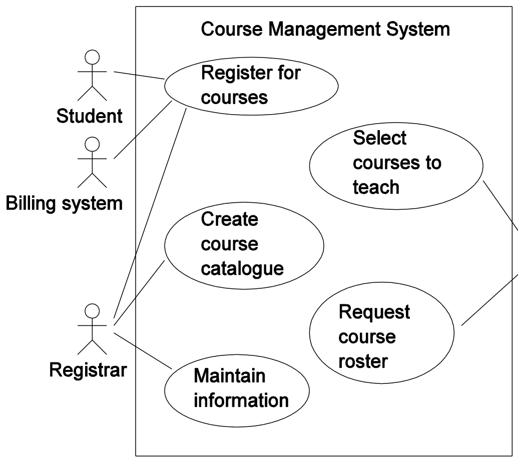

Figure 1. A sample use case diagram

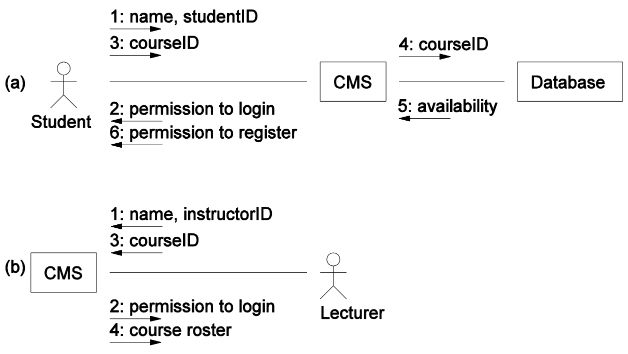

Figure 2. Two sample communication diagrams

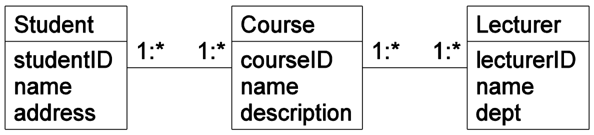

Figure 3. A sample class diagram

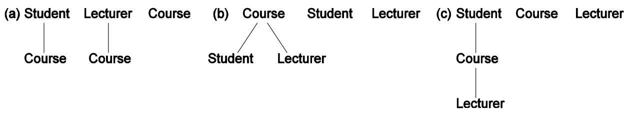

Figure 4. Several possible sets of JSON scheme trees

should be chosen. However, if the use case "request course roster" is executed more frequently, the JSON scheme trees in Figure 4b are preferred. As a comparison, Figure 4c shows a set of JSON scheme trees that will lead to redundant data, which will be shown in Figure 7.

Actual data help demonstrate the different JSON schemas. Let us assume there are three students, four courses and two lecturers for the classes in Fig. 3. Fig. 5 shows the data in JSON format that were constructed according to the JSON scheme trees in Fig. 4a. The three students are stored in an array named "students" and similarly the four courses and the two lecturers are respectively stored in the named arrays "courses" and "lecturers."

Figure 4a dictates that the registered courses of each student are clustered with the student and the courses taught by each lecturer are also clustered with the lecturer. For example, John, whose ID is S101, has registered for courses C111, C222, and C333, and Kyle, whose ID is L222, is teaching courses C333 and C444. Obviously, the JSON data in Fig. 5 favor searches on the courses for a studentID or a lecturerID. On the other hand, the JSON data of Figure 6 clearly favor the use case "request course roster" because the IDs of the students who have registered for each course are clustered with the course itself. Note that the JSON data in Figure 5 and the one in Figure 6 have no redundancy.

As a comparison, the partial JSON data in Figure 7, which were constructed according to the JSON scheme trees in Figure 4c, have redundant data. The fact that lecturer L111 is teaching courses C111 and C222 is recorded twice. This redundant data will lead to high update cost because multiple copies of the same information need to be updated together if lecturers' teaching assignments change frequently.

```
    {"students":[
       {"studentID":"S101","name":"John","address":"101 Maple street",

"courseIDs":[{"courseID":"C111"},{"courseID":"C222"},{"courseID":"C333"}]},
       {"studentID":"S202","name":"Mary","address":"202 Oak street",
         "courseIDs":[{"courseID":"C111"}]},
       {"studentID":"S303","name":"Tom","address":"303 Cedar street",
         "courseIDs":[{"courseID":"C222"},{"courseID":"C444"}]}],
     "courses":[
       {"courseID":"C111","name":"Intro   to   IS","description":"The   first
course of IS"},
       {"courseID":"C222","name":"IS  with  others","description":"The  second
course of IS"},
       {"courseID":"C333","name":"IS   with   human","description":"The   third
course of IS"},
       {"courseID":"C444","name":"Advanced IS","description":"The last course
of IS"}],
     "lecturers":[
       {"lecturerID":"L111","name":"Kevin","dept":"Math",
         "courseIDs":[{"courseID":"C111"},{"courseID":"C222"}]},
       {"lecturerID":"L222","name":"Kyle","dept":"Information Systems",
         "courseIDs":[{"courseID":"C333"},{"courseID":"C444"}]}]
    }
```

Figure 5. Sample data for the JSON scheme trees in Figure 4a

```
    {"students":[
       {"studentID":"S101","name":"John","address":"101 Maple street"},
       {"studentID":"S202","name":"Mary","address":"202 Oak street"},
       {"studentID":"S303","name":"Tom","address":"303 Cedar street"}],
     "courses":[
       {"courseID":"C111","name":"Intro   to   IS","description":"The   first
course of IS",
         "studentIDs":[{"studentID":"S101"},{"studentID":"S202"}],
             "lecturerIDs":[{"lecturerID":"L111"}]},
       {"courseID":"C222","name":"IS  with  others","description":"The  second
course of IS",
         "studentIDs":[{"studentID":"S101"},{"studentID":"S303"}],
             "lecturerIDs":[{"lecturerID":"L111"}]},
       {"courseID":"C333","name":"IS   with   human","description":"The   third
course of IS",
         "studentIDs":[{"studentID":"S101"}],
             "lecturerIDs":[{"lecturerID":"L222"}]},
       {"courseID":"C444","name":"Advanced IS","description":"The last course
of IS",
         "studentIDs":[{"studentID":"S303"}],
             "lecturerIDs":[{"lecturerID":"L222"}]}],
     "lecturers":[
       {"lecturerID":"L111","name":"Kevin","dept":"Math"},
       {"lecturerID":"L222","name":"Kyle","dept":"Information Systems"}]
    }
```

Figure 6. Sample data for the JSON scheme trees in Figure 4b

```
    {"students":[
       {"studentID":"S101","name":"John","address":"101 Maple street",
        "courseIDs":[{"courseID":"C111","lecturerID":"L111"},
                     {"courseID":"C222","lecturerID":"L111"},
                     {"courseID":"C333","lecturerID":"L222"}]},
       {"studentID":"S202","name":"Mary","address":"202 Oak street",
        "courseIDs":[{"courseID":"C111","lecturerID":"L111"}]},
       {"studentID":"S303","name":"Tom","address":"303 Cedar street",
        "courseIDs":[{"courseID":"C222","lecturerID":"L111"},
                     {"courseID":"C444","lecturerID":"L222"}]}]
    }
```

Figure 7. Partial JSON data that were constructed according to the JSON scheme trees in Figure 4c

In the remainder of the paper we shall present an algorithm to produce a good JSON schema design from UML use case diagrams, communication diagrams and class diagrams.

III. FOUNDATIONAL DEFINITIONS

UML is a well-defined modeling language (http://www.uml.org/) and JSON's definition can be found at http://www.json.org/. Thus, their definitions are not repeated here. We proceed to the other definitions for this paper.

Definition 1: Given a UML class diagram $\mathcal{C}$, a *JSON scheme tree* $\mathcal{T}$ is a tree such that each node is a class in $\mathcal{C}$.

Given the class diagram in Figure 3, all trees in Figure 4 are JSON scheme trees with respect to Definition 1. Further, each object in a UML class diagram is represented by a JSON object, which is an unordered set of name/value pairs. An object begins with { (left brace) and ends with } (right brace). Each name is followed by : (colon) and the name/value pairs are separated by , (comma) (http://www.json.org/). For example, the student John with the ID "S101" and address "101 Maple street" is represented by the JSON object `{"studentID":"S101","name":"John","address":"101 Maple street"}`. In addition, each set of objects is represented by a named JSON array of objects. For example, the set of the three students John, Mary, and Tom is represented by the named array `"students":[]`.

Definition 2: A JSON scheme tree instance $\mathcal{J}$ for a JSON scheme tree $\mathcal{T}$ is a named JSON array whose name is the plural form of the root $\mathcal{R}$ of $\mathcal{T}$. In this array each object $\mathscr{o}$ of the root $\mathcal{R}$ of $\mathcal{T}$ is represented by a JSON object $\mathscr{j}$ that is extended as follows: (1) for each child class of $\mathcal{R}$, $\mathscr{j}$ is extended by an array named by the plural form of the name of that child class and that array contains the JSON objects that represent the objects or the IDs of that child class, (2) this extension is recursively applied to each pair of parent and child nodes in $\mathcal{T}$, and (3) whenever an object of a child node is related to more than one object of a parent class, replace the objects of the child class by their IDs instead.

To illustrate Definition 2, consider the class diagram in Fig. 8a. Based on the multiplicity, the relationship between the classes A and B is many-to-many and so is the one between the classes C and D. As such, an A object may relate to many B objects and an B object may relate to many A objects. Similarly, a C object may relate to many D objects and a D object may relate to many C objects. On the other hand, because of the multiplicity 1:1, a C object can only relate to one B object but a B object may relate to many C objects. Hence the relationship between the classes B and C is one-to-many.

Given the JSON scheme trees in Figure 8b, Figure 9 shows a sample JSON scheme tree instance. The array "Bs" contains two B objects whose IDs are b101 and b202. Because an A object may relate to more than one B object, to avoid redundancy each of the B objects is clustered with its attribute x and the IDs of the related A objects. Because each C object can only relate to one B object, there are two choices to cluster the related C objects with each B object. We can cluster the related C objects or the IDs of the related C objects with each B object. In Figure 9, we chose to cluster the related C objects with each B object. Note that there is no redundancy because each C object only appears once. Because the relationship between the classes C and D is many-to-many, we only cluster the IDs of the related D objects with each C object. The other two JSON scheme trees in Figure 8b are A and D, which are degenerated JSON scheme trees because they only have the root node. They are necessary because only the IDs of A and D appear in the JSON scheme tree rooted with B and thus we need two more JSON scheme trees to contain all of the attributes of A and D. A similar argument applies to the JSON scheme tree instances in Figures 5 and 6.

Based on the relational database theory (Maier, 1983), the formal definition of Nested Normal Form can be found in (Mok et al., 1996). However, a UML class diagram does not have to satisfy all of the underlying assumptions of the relational database theory. For example, an underlying assumption of the relational database theory is that every attribute plays a unique role. This is not true for UML class diagrams because an attribute thereof may play more than one role. In Fig. 3, the attribute "name" may denote the name of the class Student or the name of the class Course. Following the same spirit of (Mok, 2007), we need to modify the definition of Nested Normal Form for this paper.

Definition 3. Given a UML class diagram $\boldsymbol{\mathcal{C}}$, a JSON scheme tree $\boldsymbol{\mathcal{T}}$ is in Nested Normal Form if (1) every class in $\boldsymbol{\mathcal{C}}$ has an attribute called ID, (2) every class in $\boldsymbol{\mathcal{C}}$ is in BCNF (Maier, 1983), (3) if $\boldsymbol{\mathcal{N}}1 \rightarrow \boldsymbol{\mathcal{N}}2$ is a nontrivial FD (functional dependency) in $\boldsymbol{\mathcal{T}}$ where $\boldsymbol{\mathcal{N}}1$ and $\boldsymbol{\mathcal{N}}2$ are two nodes in $\boldsymbol{\mathcal{T}}$, then $\boldsymbol{\mathcal{N}}1 \rightarrow \boldsymbol{\mathcal{N}}$ where $\boldsymbol{\mathcal{N}}$ is $\boldsymbol{\mathcal{N}}2$ or $\boldsymbol{\mathcal{N}}$ is a node above $\boldsymbol{\mathcal{N}}2$ in $\boldsymbol{\mathcal{T}}$, and (4) the MVDs (multivalued dependencies) implied by $\boldsymbol{\mathcal{T}}$ and the FDs that hold for $\boldsymbol{\mathcal{T}}$ is equivalent to the MVDs that hold for $\boldsymbol{\mathcal{T}}$. □

With respect to Definition 3, every JSON scheme tree in Figs. 4a and 4b is in Nested Normal Form. Every JSON scheme tree in Fig. 8b is also in Nested Normal Form. To see this, consider the nontrivial scheme tree in Fig. 8b. Note that C → B and there is no class above B in that tree. Thus, it satisfies Definition 3. However, the nontrivial JSON scheme tree in Fig. 4c is not in Nested Normal Form. To see this, note that the class diagram in Fig. 3 implies the MVD Course →→ Lecturer (a course can be taught by one or more lecturer) but this MVD cannot be implied by the tree and the FDs that hold for the tree. (There are no FDs that hold for that tree.)

IV. ALGORITHM

This section presents an algorithm that generates JSON scheme trees in Nested Normal Form from UML use case diagrams, communication diagrams, and class diagrams.

Algorithm 1

Input: a UML class diagram, a UML use case diagram, a UML communication diagram

Output: a set of JSON scheme trees in Nested Normal Form with respect to Definition 3

1. Let $\boldsymbol{u}1$, $\boldsymbol{u}2$, …, $\boldsymbol{u}$n be the given use cases in the use case diagram. Derive a communication diagram $\boldsymbol{c}$i for each use case $\boldsymbol{u}$i.

2. Let $\boldsymbol{c}1$, $\boldsymbol{c}2$, …, $\boldsymbol{c}$n be the communication diagrams for the use cases $\boldsymbol{u}1$, $\boldsymbol{u}2$, …, $\boldsymbol{u}$n respectively. Let $\boldsymbol{f}1$, $\boldsymbol{f}2$, …, $\boldsymbol{f}$n be the execution frequencies for the use cases $\boldsymbol{u}1$, $\boldsymbol{u}2$, …, $\boldsymbol{u}$n respectively. Make a list $\boldsymbol{\ell}$ of the classes in the class diagram in terms of their access frequencies.

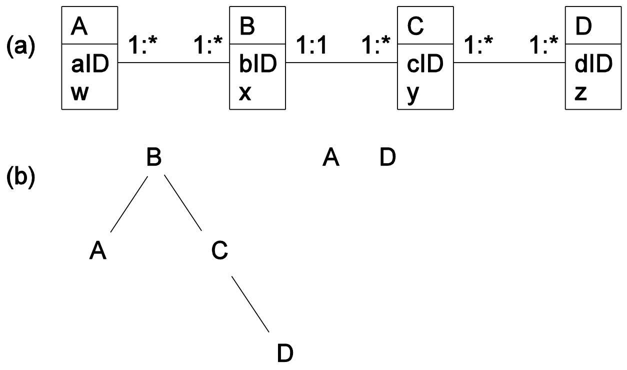

Figure 8. A sample class diagram and some sample JSON scheme trees

```
{"Bs":[
   {"bID":"b101","x":"x1",
     "aIDs":[{"aID":"a101"},{"aID":"a202"}],
         "Cs":[{"cID":"c101","y":"y1","dIDs":[{"dID":"d101"}]},

{"cID":"c202","y":"y2","dIDs":[{"dID":"d101"},{"dID":"d202"}]},

{"cID":"c303","y":"y3","dIDs":[{"dID":"d202"},{"dID":"d303"}]}]},
   {"bID":"b202","x":"x2",
     "aIDs":[{"aID":"a202"}],
         "Cs":[{"cID":"c404","y":"y4","dIDs":[{"dID":"d303"}]},

{"cID":"c505","y":"y5","dIDs":[{"dID":"d303"},{"dID":"d404"}]}]}],
  "As":[
   {"aID":"a101","w":"w1"},
   {"aID":"a202","w":"w2"}],
  "Ds":[
   {"dID":"d101","z":"z1"},
   {"dID":"d202","z":"z2"},
   {"dID":"d303","z":"z3"},
   {"dID":"d404","z":"z4"}]
 }
```

Figure 9. A sample JSON scheme tree instance for the JSON scheme trees in Figure 8a

3. Select an unmarked class $\boldsymbol{c}$ in the class diagram with the highest access frequency in $\boldsymbol{\ell}$. Make $\boldsymbol{c}$ a root node. Mark $\boldsymbol{c}$ "continued." (A node marked "continued" means that it can have child nodes.)

4. Let $\boldsymbol{n}$ be a node in a tree that is marked with "continued." For each class $\boldsymbol{c}$ in the class diagram that is connected with $\boldsymbol{n}$ with an unmarked relationship, make $\boldsymbol{c}$ a child node of $\boldsymbol{n}$ and mark the relationship "done." Further, if $\boldsymbol{c}$ has a many-to-one or one-to-one relationship with $\boldsymbol{n}$, mark $\boldsymbol{c}$ "continued" as well.

5. Repeat Steps 3 and 4 until every class in the use case diagram is marked with "continued" and every relationship with "done."

6. If a class $\boldsymbol{c}$ has more than one parent in one or more trees, replace each occurrence of $\boldsymbol{c}$ that is not a root node by the IDs of $\boldsymbol{c}$. If $\boldsymbol{c}$ is not a root node, make $\boldsymbol{c}$ the root node of a degenerated JSON scheme tree. □

We now trace through the construction of the JSON scheme trees in Fig. 4a from the use case diagram, the communication diagram, and the class diagram in Figs. 1, 2, and 3 respectively. Steps 1 and 2 of the algorithm construct the communication diagrams for the use cases. After that, we next examine the use cases' execution frequencies. In this step, the analysts and the stakeholders of the system perform an estimate based on their past experiences. Suppose that "register for courses" is the most frequently executed use case. Hence, the JSON data format should be designed with this use case in mind. Thus, Step 3 of the algorithm chooses the class Student as the root of a JSON scheme tree. The class Student is also marked with “continued,” signaling that it can have child nodes. In the class diagram in Figure 3, the class Course is connected to the class Student through a many-to-many relationship. Hence, the class Course is added as a child node to the class Student. However, it is not marked with “continued” because the relationship between them is many-to-many. The result is one of the trees in Figure 4a. Suppose that the class Course is the next frequently access class. It is then selected as the root of another JSON scheme tree. The class Lecturer, but not the class Student, is then added as a child node to Course. The class Student is not added because the relationship between the classes Student and Course has already been marked with “done.” Finally, the class Lecturer is made the root of a JSON scheme tree. However, the class Course is not added as a child node to Lecturer because the relationship between Course and Lecturer has already been marked with “done.”

V. CONCLUSIONS

This paper presents a JSON scheme tree generation algorithm that generates JSON scheme trees in Nested Normal Form. As a result, the generated JSON scheme trees are redundancy-free. As our future work, we shall devise experiments to actually measure the transfer time and query time against JSON schemas with or without redundancy to empirically show that JSON schemas in Nested Normal Form have advantages over those that do not satisfy Nested Normal Form.

REFERENCES

[1] Maier, D. (1983). The Theory of Relational Databases, Computer Science Press, Rockville, Maryland, USA.

[2] Mok, W. Y. (2007). Designing nesting structures of user-defined types in object-relational databases. Information & Software Technology, 49(9/10), 1017-1029. https://doi.org/10.1016/j.infsof.2006.10.008

[3] Mok, W. Y., Ng, Y., & Embley, D. W. (1996). A normal form for precisely characterizing redundancy in nested relations. ACM Transactions On Database Systems, 21(1), 77-106. https://doi.org/10.1145/227604.227612

AUTHOR

Wai Yin Mok is with The University of Alabama in Huntsville, Huntsville, AL 35899 (mokw@uah.edu).

Submitted 03 November 2016. Published as resubmitted by the author 10 December 2016.