Article

JIS

Journal of Information Science
2015, Vol. 41(3) 383–398

DOI: 10.1177/0165551515576478
jis.sagepub.com
SAGE

A data-driven dynamic ontology

Dhomas Hatta Fudholi
La Trobe University, Australia and Universitas Islam Indonesia, Indonesia

Wenny Rahayu
La Trobe University, Australia

Eric Pardede
La Trobe University, Australia

Abstract

Valuable knowledge in every community is changed frequently. It often remains closely inside a community, even though it has huge potential to promote problem-solving in the wider community. Our research aims to increase the capability of communities in capturing, sharing and maintaining knowledge from any domain. We utilize an ontology, a shareable form, to collect, consolidate and find commonality inside knowledge. Most ontologies available these days were created by domain experts to fulfill certain domain requirements. However, in cases when domain experts are not obtainable or standard agreement within the domain is not available, such as in natural or herbal therapy domain, we propose that an ontology can also be extracted from existing knowledge-bases residing within the community. In order to achieve our aim, we design a data-driven dynamic ontology model. Our model consists of base knowledge creation and knowledge propagation phases. In the base knowledge creation phase, we define a general concept of capturing community knowledge from data into an ontology representation, rather than just transforming a specific data format into an ontology as found in existing studies. In our knowledge propagation phase, the dynamic community knowledge sources become the trigger of propagation. This is different from some approaches in existing studies, where the triggering event is an individual change inside the ontology and external data may not be the base source of the knowledge in the evolving ontology. We define the propagation feature with a novel *delta* script. The script is *minimum* yet *complete* to simplify and save knowledge sharing transportation resources. The evaluation result shows that the data-driven dynamic ontology with its propagation method not only delivers *complete* and *correct* semantics but also shows good performance in terms of operation cost and processing time.

Keywords

data-driven; dynamic; ontology; propagation

1. Introduction

Every community in every nation has unique knowledge that is valuable to share. The dormant knowledge inside each community drives a lack of global standards and common understanding. The need to share community knowledge can be facilitated through the development of a systematic method of knowledge capture, development and maintenance. This solution faces two challenges: (a) developing a shareable base knowledge from captured community knowledge representation; and (b) maintaining dynamic knowledge that changes over time while ensuring information preservation and a cost effective maintenance mechanism.

Ontologies are commonly used as good mechanisms to share rich knowledge and common understanding [1]. The knowledge building in the form of an ontology corresponds to the first challenge in developing shareable base knowledge from communities. Research has defined concepts in transforming data such as text [2–4], XML (eXtensible Markup

Corresponding author:
Dhomas Hatta Fudholi, Department of Computer Science and Information Technology, La Trobe University, Victoria 3086, Australia.
Email: dfudholi@students.latrobe.edu.au; hatta.fudholi@yahoo.co.id

Language) [5–9] and relational databases [10–12] into ontologies. While this work stresses the transformation of a specific data format into an ontology, we define a general concept of capturing knowledge from community data representation to enable base knowledge creation in the form of an ontology from a multiformat data source.

The second challenge in maintaining dynamic knowledge cannot be separated from evolution methodology. Since we use an ontology as knowledge representation, we need to follow an ontology evolution concept that keeps the ontology up to date with respect to changes in the domain that it models [13]. Ontology evolution maintains the ontology in the *fresh* state. Stojanofic et al. [14] started the ontology evolution process by capturing change, representing change, analysing inconsistencies, implementing change, propagating change and finally validating change in the ontology. The source of evolution may come from different things such as: an ontology version change log as in Sari et al. [15]; instances of information inside the ontology as in Stojanofic et al. [14]; or external data which may not be a knowledge source of the ontology as in Zablith et al. [16]. In our approach, the evolution source comes from the evolving data sources that build the base knowledge. Moreover, our focus is in the process of propagating changes to maintain the newness of the shareable knowledge in the form of an ontology. Propagation is useful to maintain the ontology when the data source is not present in the same location as the ontology. It eliminates the need to send the original file, which can be costly, especially if the file size is very big. In order to do the propagation, we define the possible changes derived from the data and use a change detection tool to create a list of occurred changes. The list of occurred changes is formed into a novel formal *delta* script.

In order to address the two challenges, we designed a general model to develop community shareable knowledge that evolves, called a data-driven dynamic ontology. The model incorporates an ontology as the shareable format and takes data representation as the community knowledge. As the proof of concept, we take evolving XML data as the community knowledge representation and analyse it in our system. We create a web-based application to evaluate our concept, especially in terms of the correctness and the effectiveness of dynamic ontology development and maintenance using propagation features against pure ontology reproduction.

Following this section, the rest of this paper is organized as follows. Section 2 gives an overview of the data-driven dynamic ontology concept. Section 3 explains the first part of the concept, which is base ontology development. Section 4 elaborates the second part of the concept, which is dynamic ontology propagation, including the design of the novel *delta* script. Section 5 presents the analytical study of the concept. The evaluation of the *completeness*, the *correctness* and the *performance* for the propagation method by means of operation cost and processing time is also delivered in this section. Finally, the conclusion and future work is described in Section 6.

2. Concept and model

The proposed data-driven dynamic ontology concept takes community knowledge as the data source. We define the minimum requirements of the knowledge data representation. We base the requirements on the way an ontology stores semantic knowledge. An ontology comprises the ontology model and ontology individuals, which may refer to schema and records in the structural data. Moreover, each ontology component, including the individual, has a unique identity. Therefore, to be able to capture the knowledge inside a community, the schema should be extractable from its raw data and the data instances should at least record a unique identity. A schema can be derived from semi-structured data (e.g. XML) and structured data (e.g. relational database). Even though there are studies to extract knowledge from unstructured text [2–4], the mining result cannot be ascertained since different user interaction inputs may reflect different outcomes of structure extraction in terms of depth, concept name and relations. In our proposed concept, we assume that there exists a semi-structured or a structured data repository within the community, in order to build the ontology model. However, instances in the ontology can be derived from unstructured data, such as unstructured data in NoSQL environments and text-based documents.

Recalling our research aims, we designed a data-driven dynamic ontology as a model of dynamic shareable knowledge development from communities. There are two key points that underpin the nature of the extracted knowledge: *shareable* and *dynamic*. The data-driven dynamic ontology model is subsequently divided into two main parts to respond to the two key points: (a) base ontology creation; and (b) ontology propagation.

The creation of the base ontology addresses the need for a uniform mechanism to describe the extracted shareable knowledge. In our model, the creation of the base ontology occurs only once. We extract knowledge of the data into the base ontology model via its schema and populate the instance. This concept is detailed in Section 3. Schema extraction from the data is preprocessed using existing tools, such as *Trang*[1] for XML Schema (XSD) extraction, or *SchemaSpy*[2] for database schema.

Once created, the ontology is continuously updated to reflect the dynamic nature of community knowledge. Ontology propagation enables the dynamic feature in knowledge development. The propagation updates the base ontology when

Table 1. General ontology extraction from data.

Data	*iSchema*	Ontology model
Structure		
An abstraction of objects collection, with common characteristics (or class)	<iClass>	owl:Class
Association relationship between classes	<iARel>	owl:ObjectProperty
Object's attributes, with literal data type	<iAttrib>	owl:DatatypeProperty
Inheritance relationship between classes	<iIRel>	rdfs:subClassOf
Object's or attribute's minimum occurrence	<iMin>	owl:minCardinality
Object's or attribute's maximum occurrence	<iMax>	owl:maxCardinality
Attribute's data type	<iType>	rdf:datatype
Instance		
Instance identity	<iD>	
Attribute's value	<iVal>	

changes in the data source occur. The old data source and the current data source are compared along with their extracted schema using the differencing processor and result in a *delta* script. We create a novel *delta* script representation that consists of the data instance and schema change lists. This list acts as a map of where the propagation should be applied and the updated knowledge that needs to be propagated. The propagation process is repeated over time when a change occurs in the data source. The detail of this part is explained in Section 4.

3. Base ontology creation

The creation of a base ontology is the first process in the data-driven dynamic ontology concept. In order to do so, we extract knowledge from the community data into the base ontology. Since knowledge data has two components of information, which are structure and instance, the base ontology creation phase has two parts: (a) structure extraction; and (b) instance population. For structure extraction, we classify the extraction based on ontology model components. We categorize the structure extraction into *class, property* and *constraint* extraction. We define the knowledge extraction concept in a general declarative representation to enable the future inclusion of any format of the knowledge source in our framework. In addition, we define the formal model using our novel representation called *iSchema*. *iSchema* is a tag-based definition for our data component knowledge representation. *iSchema* is very useful for identifying particular knowledge information for our novel *delta* script definition in Section 4. A summary of the definition is given in Table 1 and is elaborated in the following subsection. In addition, we give an illustration in Figure 1 and we refer to the literature [5–12] for examples of ontology transformation from a specific data format that comply with our knowledge extraction concept. Each phase is described below.

3.1. Class extraction

An ontology class consists of individuals that have the same characteristics. Hence, a collection of objects that has common characteristics and a class in the data is formed into an ontology class. Let <iClass> be the *iSchema* representation of the data class. In Figure 1, <iClass> refers to *Food, Supplier, Meat* and *Cereal*. We can find a collection of objects as a *complexType* in XML or a table (relation) in a relational database.

An ontology may relate one class to another in a subclass relation. A subclass relation in the ontology model can be derived from the inheritance relationship between classes. This inheritance relationship is defined as <iIRel> in *iSchema*. The definition of <iIRel> is specified by (1).

$$< \text{iIRel} > IR : \ < \text{iClass} > C_1 \rightarrow \ < \text{iClass} > C_2 \quad (1)$$

<iIRel> exist in the relationship of *Meat* and *Cereal* with *Food* in Figure 1. An extension in XML and a subtable in the database are further examples of the inheritance relationship.

3.2. Property extraction

An ontology has two kinds of properties: ObjectProperty and DatatypeProperty. ObjectProperty relates two classes in an association relationship. In data structure views, we could say that, if there exists an association relationship between

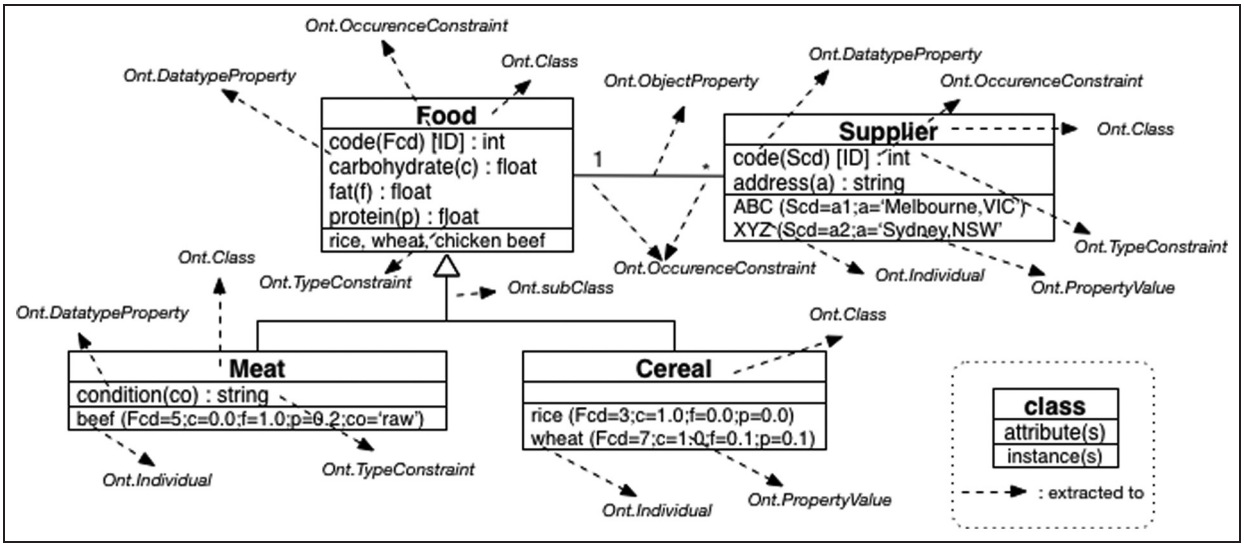

Figure 1. Knowledge extraction illustration.

objects or classes, we form the relationship into ObjectProperty. Formally, let <iARel> be the *iSchema* representation of the association relationship between classes and its definition is specified by eqn (2).

$$< \text{iARel} > AR : \ < \text{iClass} > C_1 \rightarrow \ < \text{iClass} > C_2 \tag{2}$$

<iArel> may exist in the database table relationship and XML *complexType*. In Figure 1, the association relationship is illustrated as the relationship between *Food* and *Supplier*. In addition, we need to consider the semantics of the ObjectProperty relationship. The domain and the range of particular ObjectProperty should be logical. As for the illustration, if the property that is made between *Food* and *Supplier* is called *isSuppliedBy*, the logical domain and range of the property will be *Food* and *Supplier*, respectively. On the other hand, if the property is called *hasProduct/supplies*, then the domain and range will be the opposite. We could have both property definitions in the ontology representation, where one property is the inverse property of the other.

DatatypeProperty links classes with their literal values. In data representation, it refers to the attribute that describes an object. Let <iAttrib> be the *iSchema* representation for this relationship and <iType> be the data type of the literal value. The definition of <iAttrib> is specified by eqn (3):

$$< \text{iAttrib} > A : \ < \text{iClass} > C \rightarrow \ < \text{iType} > T \tag{3}$$

<iAttrib> can be seen as XML *simpleType* and the database column, since they generally contain literal values. As illustrated in Figure 1, *code, carbohydrate, fat, protein, condition* and *address* describe a collection of objects and also have literal values. Therefore, they are extracted into DatatypeProperty.

3.3. Constraint extraction

Semantic constraints exist in every ontology. We extract two ontology constraints from the data: occurrence and data type constraints. Occurrence constraints in data reflect the minimum and maximum cardinality in an ontology. To represent the occurrence constraint in *iSchema*, we define <iMin> and <iMax> for minimum and maximum occurrences, respectively. Both occurrence constraints adhere to <iAttrib> and <iARel>. The definitions of <iAttrib> and <iARel> are then specified with occurrence constraints into eqns (4) and (5):

$$< \text{iARel} > AR[< \text{iMin} > i, \ < \text{iMax} > j] : \ < \text{iClass} > C_1 \rightarrow \ < \text{iClass} > C_2 \tag{4}$$

$$< \text{iAttrib} > A[< \text{iMin} > i, \ < \text{iMax} > j] : \ < \text{iClass} > C \rightarrow \ < \text{iType} > T \tag{5}$$

In Figure 1, the association relationship between *Food* and *Supplier* illustrates the occurrence constraint where one kind of food may have more than one supplier. Moreover, *code* is an identity (ID) that naturally needs to have exactly

one unique value. An occurrence constraint in XML is defined in *minOccurs* and *maxOccurs* attributes, while in a relational database the constraint may exist in the NOT NULL and primary key assertion.

A data type constraint adheres to data attributes in order to set the domain value. In *iSchema*, it reflects <iType>. Columns in a relational database and XML elements have certain data types to restrain its literal type of value. This constraint refers to the attribute type illustrated in Figure 1, such as float and string.

3.4. Instance population

Instance population is executed after the base ontology model has been built from data source structure extraction. Every instance in the data source is populated as an individual into the base ontology model. As an illustration from Figure 1, *rice* instance in the *Cereal* class has *carbohydrate* attribute with a value of *1.0. rice* is then populated as an ontology individual, and all the knowledge that describes *rice* is extracted in the form of a triplet statement, e.g. *rice-hasCarbohydrate-1.0*. In *iSchema*, the value of the attribute is stored as <iVal>. The unique ID record in the data source becomes the individual's ID in the ontology. In *iSchema*, the ID of the instance is stored as <iD>.

4. Dynamic ontology propagation

In this section, we elaborate the dynamic ontology propagation process. We start by presenting the nature of the changes in the data as the cause of the propagation. We follow this by explaining the need for differencing between the current data source and its previous versions. Finally, we describe the novel *delta* script forms.

4.1. Changes in data

The knowledge inside a community changes and is updated frequently. We focus on data source change, not in its schema, since initially schema may not be available. Moreover, we see semantic changes in data rather than its syntactical representation change as in Guerrini et al. [17], which differs in the global and local declaration.

We identify the changes in data as the basic edit operations of insert, delete, update and move. The update operation includes changes in the name, value, type and constraint. We use the Selkow's tree model [18] interpretation for the delete operation. Selkow's model may be directly used in an XML-based data tree. However, we could take a general definition where an object is deleted, and the entire definition (inherited object, attributes) rooted at the particular object is also deleted.

4.2. Differencing

To identify evolving knowledge, we use a differencing process, which is an important process during propagation. While differencing is used to show the part of knowledge that needs to be propagated, it also has additional features as in Cobéna et al. [19]: version and querying past knowledge; learning about changes; monitoring changes; and indexing. An ontology is formed from the model and individuals, which reflect the structure and instances. Hence, differencing should be able to identify the knowledge changes (delete, insert, update and moved) inside the structure and also the instances. Referring to our knowledge structure representation *iSchema*, differencing should detect all changes in <iClass>, <iARel>, <iAttrib>, <iIRel>, <iMin>, <iMax> and <iType>.

X-Diff [20] and *XANDY* [21] are among the data instance differencing tools. Both are created to be high-performance tools with the capability of comparing an unordered XML data tree. *X-Diff* uses an effective algorithm that integrates key XML structure characteristics with standard tree-to-tree correction techniques [20]. On the other hand, *XANDY* uses a relational database that converts XML documents into relational tuples and detects the changes in these tuples using SQL queries. *XANDY* is claimed to hold better scalability compared with *X-Diff* and is capable of detecting inserted nodes, deleted nodes and updated nodes [21]. For differencing in structure, the most recent work and the one that we will adopt is *XS-Diff* [22]. *XS-Diff* stores XSD versions in a relational database. Thus, the change detection results and the *delta* information are employed in relational tables [22]. *XS-Diff* is proven to be complete, semantically correct and an optimal tool. *XS-Diff* can detect the deletion, insertion, update and migration/move of XSD elements and attributes. The update detection of the elements or attributes includes the type and occurrence update. *XANDY* and *XS-Diff* use a relational database as the mediation; therefore, their concept is applicable for detecting relational database data differences.

We take the ideas of *XANDY* and *XS-Diff* in the differencing concept for instance and structure changes, respectively. Both tools deliver outstanding results in detecting changes. However, in the structure part, they cannot detect an update in structure name (rename) by itself. To enable this feature, we combine the capability in both differencing concepts. A

renamed attribute might occur when the instance differencing part detects a massive attribute deletion and insertion in the same path but holding the same value, and the structure differencing part detects a deletion and insertion of an attribute in the same parent path where the information on the attribute name is the same as in the instance detection part.

4.3. Delta script

Changes of data are collected in a *delta* script. The *delta* script is used as a tool to maintain the knowledge representation in an ontology. A *delta* script is very useful when the original file is located in another place or in the distributed environment, since sending the whole updated file will consume resources and result in a greater chance of information loss.

Cobéna et al. [23] listed several important features that tools should possess in order to generate a *delta* file. These are *completeness, minimality, performance and complexity, 'move' operation* and *semantics*. We adopt these as indicators of a good *delta* script. A good *delta* script should list sufficient information to transform the old version of knowledge to the current one (*complete*), lists a concise sequence of information to save storage space and network bandwidth (*minimality*), delivers good scalable performance with low memory usage (*performance and complexity*), is capable of detecting 'move' operations to achieve better performance and minimality (*'move' operation*), and contains precise semantics information based on the data structure (*semantics*).

In this paper, we propose a real-time propagation that updates the old ontology to the current version when there is a change in the data source. We do not store the previous *delta* script, which could be used as a back-propagation tool. This concept will be investigated in future work. By taking the aforementioned important features into consideration, we create a definition of our own set of *delta* script representations. There are two groups of script in the *delta* script that contain the differences in the instances and the differences in the structure.

As a case study to prove our proposed concept, we use part of the Food Display Table[3] XML dataset from the USDA (United States Department of Agriculture) and make updates to some of the data. Figure 2 shows the original part of the dataset and the updated dataset. The bold characters indicate the changed part of the dataset. Figure 3 presents the original XSD from the dataset against the updated XSD. Both XSD are produced by *Trang* engine in *oXygen*[4] XML Editor.

The set of the *delta* script is defined as follows:

Original XML Dataset

```
<Food_Display_Table><Food_Display_Row>
  <Food_Code>13110100</Food_Code>
  <Display_Name>Ice cream, regular</Display_Name>
  <Portion_Display_Name>cup</Portion_Display_Name>
 <Grains>.00000</Grains><Whole_Grains>.00000</Whole_Grains>
  <Vegetables>.00000</Vegetables>
  <Orange_Vegetables>.00000</Orange_Vegetables>
  <Drkgreen_Vegetables>.00000</Drkgreen_Vegetables>
  <Starchy_vegetables>.00000</Starchy_vegetables>
  <Other_Vegetables>.00000</Other_Vegetables>
  <Oils>.00000</Oils>
 </Food_Display_Row><Food_Display_Row>
  <Food_Code>13110120</Food_Code>
  <Display_Name>Ice cream, rich</Display_Name>
  <Portion_Display_Name>cup</Portion_Display_Name>
 <Grains>.00000</Grains><Whole_Grains>.00000</Whole_Grains>
  <Vegetables>.00000</Vegetables>
  <Orange_Vegetables>.00000</Orange_Vegetables>
  <Drkgreen_Vegetables>.00000</Drkgreen_Vegetables>
  <Starchy_vegetables>.00000</Starchy_vegetables>
  <Other_Vegetables>.00000</Other_Vegetables>
<Oils>.00000</Oils></Food_Display_Row></Food_Display_Table>
```

Updated XML Dataset

```
<Food_Display_Table><Food_Display_Row>
  <Food_Code>13110100</Food_Code>
  <Display_Name lang="en">Ice cream, regular</Display_Name>
  <Portion_Display_Name>cup</Portion_Display_Name>
  <Whole_Grains>.00000</Whole_Grains>
  <Vegetables_Total>.00000</Vegetables_Total><Vegetable_Detail>
   <Orange_Vegetables>.00000</Orange_Vegetables>
   <Drkgreen_Vegetables>.00000</Drkgreen_Vegetables>
   <Starchy_vegetables>.00000</Starchy_vegetables>
   <Other_Vegetables>.00000</Other_Vegetables></Vegetable_Detail>
  <Oils>.00000</Oils></Food_Display_Row><Food_Display_Row>
  <Food_Code>13110120</Food_Code>
  <Display_Name lang="en">Ice cream, rich</Display_Name>
  <Portion_Display_Name>cup or scoop</Portion_Display_Name>
  <Whole_Grains>.00000</Whole_Grains>
  <Vegetables_Total>.00000</Vegetables_Total>
  <Vegetable_Detail>
   <Orange_Vegetables>.00000</Orange_Vegetables>
   <Drkgreen_Vegetables>.00000</Drkgreen_Vegetables>
   <Starchy_vegetables>.00000</Starchy_vegetables>
   <Other_Vegetables>.00000</Other_Vegetables>
  </Vegetable_Detail></Food_Display_Row></Food_Display_Table>
```

Figure 2. Case study: part of the USDA Food Display Table Database (original and updated).

Original XSD

```
<xs:element name="Food_Display_Table">
 <xs:complexType><xs:sequence>
  <xs:element maxOccurs="unbounded" ref="Food_Display_Row"/>
   </xs:sequence></xs:complexType></xs:element>
  <xs:element name="Food_Display_Row">
  <xs:complexType><xs:sequence><xs:element ref="Food_Code"/>
   <xs:element ref="Display_Name"/>
   <xs:element ref="Portion_Display_Name"/>
   <xs:element ref="Grains"/><xs:element ref="Whole_Grains"/>
   <xs:element ref="Vegetables"/>
   <xs:element ref="Orange_Vegetables"/>
   <xs:element ref="Drkgreen_Vegetables"/>
   <xs:element ref="Starchy_vegetables"/>
   <xs:element ref="Other_Vegetables"/>
   <xs:element ref="Oils"/>
  </xs:sequence></xs:complexType></xs:element>
  <xs:element name="Food_Code" type="xs:integer"/>
  <xs:element name="Display_Name" type="xs:string"/>
  <xs:element name="Portion_Display_Name" type="xs:NCName"/>
  <xs:element name="Grains" type="xs:decimal"/>
  <xs:element name="Whole_Grains" type="xs:decimal"/>
  <xs:element name="Vegetables" type="xs:decimal"/>
```

Updated XSD

```
<xs:element name="Food_Display_Table">
 <xs:complexType><xs:sequence>
  <xs:element maxOccurs="unbounded" ref="Food_Display_Row"/>
  </xs:sequence></xs:complexType></xs:element>
  <xs:element name="Food_Display_Row"><xs:complexType><xs:sequence>
  <xs:element ref="Food_Code"/><xs:element ref="Display_Name"/>
<xs:element ref="Portion_Display_Name"/><xs:element
ref="Whole_Grains"/>
<xs:element ref="Vegetables_Total"/><xs:element
ref="Vegetable_Detail"/>
<xs:element minOccurs="0" ref="Oils"/>
  </xs:sequence></xs:complexType></xs:element>
  <xs:element name="Food_Code" type="xs:integer"/>
  <xs:element name="Display_Name"><xs:complexType mixed="true">
<xs:attribute name="lang" use="required" type="xs:NCName"/>
   </xs:complexType></xs:element>
  <xs:element name="Portion_Display_Name" type="xs:string"/>
  <xs:element name="Whole_Grains" type="xs:decimal"/>
  <xs:element name="Vegetables_Total" type="xs:decimal"/>
  <xs:element name="Vegetable_Detail"><xs:complexType><xs:sequence>
    <xs:element ref="Orange_Vegetables"/>
    <xs:element ref="Drkgreen_Vegetables"/>
    <xs:element ref="Starchy_vegetables"/>
    <xs:element ref="Other_Vegetables"/>
    </xs:sequence></xs:complexType></xs:element>
```

Figure 3. Case study: part of the USDA Food Display Table Dataset's XSD (original and updated).

Definition 1. $\Delta \equiv <\boldsymbol{D}, \boldsymbol{I}, \boldsymbol{U}, \boldsymbol{M}, ID>$. *Delta* script Δ comprises four sets of lists, which are *delete (**D**), insert (**I**), update (**U**)* and *move (**M**)*, and an attribute path and/or name that holds the unique ID (*ID*). These lists are proposed to achieve *completeness* and *minimality* yet hold solid *semantics*. The *update* list and *move* list have been designed to minimize the primitive operation of delete and insert. In addition, each list is compiled from the minimum information tuple to proceed into a complete propagation. The location of the ID is predefined. The *ID* value of the instance cannot be updated. If it is updated, then it will drive a delete and insert process for the whole data instance block. This restriction is applied since the *ID* is used as the location information by every changed instance member, and it needs to remain the same and unique.

The sequence of the listed differences in the *delta* script is (***D***→***I***→***U***→***M***). The first list is the delete list. The delete process should come first to avoid possible name duplication of the new object or property. The second list is the insert list. The third list is the update list. Finally, the fourth list is the move list. The move list should be the last list because we need to state all of the inserted and updated components before we are able to identify the possible target of the moved component. Since there is a possibility of having a difference list of both instance and structure, the structure list goes before the instance in each list. The modification in structure should be applied before the instance.

Definition 2. ***Delete List D*** is defined as $\boldsymbol{D} \equiv <\boldsymbol{SDel}, \boldsymbol{IDel}>$. ***D*** consists of a set of deleted structure item(s) (***SDel***) and deleted instance item(s) (***IDel***). ***SDel*** is a collection of deleted <iClass> and/or <iAttrib>. ***IDel*** is a collection of deleted instances, which is defined as a tuple of (<iD>, <iAttrib>, <iVal>). <iD> is the identity value of an instance. <iVal> is the value in <iAttrib>. The property value in the deleted instance needs to be listed. This is done to overcome the conflict when there is more than one element alike in the instances with different values.

There are three rules associated with **Definition 2**. (a) When deleting an <iAttrib> of an instance, the value inside it will also be deleted. (b) The deletion of <iClass> will also remove its entire child nodes and any of its associated relationships. By keeping *minimality* in mind, the only listed child nodes in the *delta* script are <iClass> and <iAttrib>. Any related associated relationship can be derived from deleted <iClass> and does not need to be listed. (c) When there

is <iClass> and/or <iAttrib> structure deletion, all respective instance data will be deleted. Since the number of instances may be very large, the respective deleted instances do not need to be listed in the *delta* script.

Three delete operations occur in the data in our case study: (a) the deletion of the 'Oils' element in the data instance; (b) the deletion of the 'Grains' structure; and (c) the deletion of the 'Display_Name' structure. From **Definition 2**, we can write the *delta* as follows:

```
#Delete Structure
<iAttrib>Grains
<iAttrib>Display_Name
#Delete Instance
<iD>13110120<iAttrib>Oils<iVal>.00000
```

Definition 3. ***Insert List I*** is defined as $I \equiv$ <***SIns, IIns***>. ***I*** has a set of inserted structure item(s) (***SIns***) and inserted instance item(s) (***IIns***). ***SIns*** is a collection of tuples (<iClass>,(<iARel> OR <iIRel>),<iMin>,<iMax>) for class and (<iAttrib>, <iClass>, <iType>, <iMin>, <iMax>) for attribute. The value of <iArel> and <iIRel> in class insertion tuple is the other classes related to the inserted class. <iClass> and <iType> in the attribute insertion tuple refer to the class that owns the attribute and the data type of the attribute, respectively. ***IIns*** is a collection of tuples (<iD>, <iAttrib>, <iClass>, <iVal>). <iClass> instance insertion tuple refers to the class that owns the attribute.

In the case study, there exist some insertion operations. A new element called 'Vegetable_Detail' is added in the structure. A new attribute 'lang' is added to 'Display_Name'. In this case, the old attribute 'Display_Name' is deleted and a new element and attribute 'Display_Name' is created. The new attributes 'lang' and 'Display_Name' have a value each in each data instance. From **Definition 3**, we may write the *delta* script as follows:

```
#Insert Structure
<iClass>Vegetable_Detail<iARel>Food_Display_Row<iMin>1<iMax>1
<iClass>Display_Name<iARel>Food_Display_Row<iMin>1<iMax>1
<iAttrib>Display_Name<iClass>Display_Name<iType>ANY<iMin>1<iMax>1
<iAttrib>lang<iClass>Display_Name<iType>xs:NCName<iMin>1<iMax>1
#Insert Instance
<iD>13110100<iAttrib>Display_Name<iClass>Display_Name<iVal>"Icecream, regular"
<iD>13110120<iAttrib>Display_Name<iClass>Display_Name<iVal>"Ice cream, rich"
<iD>13110100<iAttrib>lang<iClass>Display_Name<iVal>"en"
<iD>13110120<iAttrib>lang<iClass>Display_Name<iVal>"en"
```

Definition 4. ***Update List U*** can be described as $U \equiv$ <***SUpd, IUpd***>. ***U*** consists of a set of updated structure item(s) (***SUpd***) and updated instance item(s) (***IUpd***). Updates in structure may include renaming, changing the data type and changing the occurrence. On the other hand, instance updates include only changing the value of the attribute. ***SUpd*** is a collection of tuples (*old*(<iClass>), *new*(<iClass>), (<iARel> OR <iIRel>), <iMin>, <iMax>) for class, and (*old*(<iAttrib>), *new*(<iAttrib>), <iClass>, <iType>, <iMin>, <iMax>) for attribute. <iClass> in the attribute update tuple is the described class of respective attributes and the <iType> is the new data type of the attribute. **IUpd** is collection of tuples (<iD>, <iAttrib>, <iClass>, *old*<iVal>, new<iVal>). <iClass> in the instance update tuple is the described class of the attribute.

In the case study, the element 'Vegetables' is renamed 'Vegetables_Total', the minimum occurrence of element 'Oils' is updated to 0 and the data type of 'Portion_Display_Name' is changed. From **Definition 4**, we can write the *delta* script as follows:

```
#Update Structure
<iAttrib>Vegetables<iAttrib>Vegetables_Total<iClass>Food_
Display_Row<iType>xs:decimal<iMin>1<iMax>1
<iAttrib>Oils<iAttrib>Oils<iClass>Food_Display_Row<iType>xs:decimal <iMin>0<iMax>1
```

```
<iAttrib>Portion_Display_Name<iAttrib>Portion_Display_Name<iClass>
Food_Display_Row<iType>xs:string<iMin>1<iMax>1
#Update Instance
<iD>13110120<iAttrib>Portion_Display_Name<iClass>Food_Display_Row<iVal>"cup"<iVal>"c-
up or scoop"
```

Definition 5. ***Moved List M*** is defined as $M \equiv SMov$. ***M*** consists of a set of moved structure item(s) (***SMov***). ***SMov*** is a collection of tuples (<iClass>, *old*<iClass>, *new*<iClass>) for class, and (<iAttrib>, *old*<iClass>, *new*<iClass>) for attribute. *old*<iClass> and *new*<iClass> refer to the old and the new superclasses for a class move operation. As for the attribute move operation, it refers to the old and the new class that is described.

In the case study, there are four move operations, which are the movement of 'Orange_Vegetables', 'Drkgreen_Vegetables', 'Starchy Vegetables' and 'Other_Vegetables' attributes under the 'Vegetable_Detail' element. From **Definition 5**, we can write the *delta* script as follows:

```
#Move Structure
<iAttrib>Orange_Vegetables<iClass>Food_Display_Row<iClass>Vegetable_Detail
<iAttrib>Drkgreen_Vegetables<iClass>Food_Display_Row<iClass>Vegetable_Detail
<iAttrib>Starchy_Vegetables<iClass>Food_Display_Row<iClass>Vegetable_Detail
<iAttrib>Other_Vegetables<iClass>Food_Display_Row<iClass>Vegetable_Detail
```

4.4. Propagation

Ontology propagation uses the *delta* script to identify knowledge changes. The changes may occur in the ontology model as its structure and/or in ontology individuals as its instances.

In terms of propagating the ontology model, knowledge representation *iSchema* inside the *delta* script refers to a specific part of the ontology model component, as shown in Table 1. Accordingly, any operation for *iSchema* representation will directly propagate the referring ontology model component. However, we do need to consider the semantics of the propagated part. The definition of *delta* script has been shaped into a form that takes the *semantic* factor into consideration. To clarify the propagation process, we elaborate four propagations processes from the *delta* script listed item by showing their derived knowledge and changes in the OWL (Web Ontology Language) representation. In our framework implementation, ObjectProperty and DatatypeProperty name are prefixed by 'has' and 'dp', respectively. Table 2 shows a list of four changes in the case study, namely delete, insert, update and move operation. The OWL representation part in Table 2 shows the implementation of the OWL ontology, where a ~~strikethrough~~ word means 'deleted' and '>>>' symbol means 'of' or 'which is connected to'.

The instance of a class in OWL is referred to as an individual, each of which is a resource. The set of individuals in a class is considered as its *class extension* [24]. To undertake instance propagation in the OWL ontology, we have to manage it in the form of a triplet statement (subject, predicate and object). The subject of the statement is the individual itself. The predicate of the statement is the property of the individual. Finally, the object is the literal value of the property in the respective individual. As an example, we elaborate three changes, that is, delete, insert and update the *delta* script listed items from a case study in Table 3, where a ~~strikethrough~~ words means 'deleted'.

5. Evaluation and discussion

In this section, we evaluate the proposed mechanism for facilitating the dynamic notion of ontology. We aim to show that the data-driven dynamic ontology with its propagation feature offers correct knowledge preservation and gives better performance in term of resources. Our evaluation considers three factors, namely, *completeness, correctness* and *performance*. As the proof of our concept, we create an application to evaluate the case study using Jena,[5] a Java framework for building Semantic Web applications.

Table 2. Ontology model propagation case study.

Delta script	Derived knowledge	OWL repesentation
Delete		
<iAttrib>Grains	An attribute 'Grain' is deleted. Hence, 'dpGrain' ontology DatatypeProperty is deleted.	~~owl:DatatypeProperty 'dpGrain'~~
	Constraint information of 'dpGrain' is deleted.	~~owl:Restriction >>> owl:onProperty >>> owl:DatatypeProperty 'dpGrain', owl:minCardinality, owl:maxCardinality 1~~
	All 'dpGrain' knowledge and value in all individual is deleted.	~~<dpGrain rdf:datatype="xs:decimal"> * </dpGrain>~~
Insert		
<iClass>Vegetable_Detail <iARel>Food_Display_Row <iMin>1<iMax>1	A new ontology Class 'Vegetable_Detail' is created.	owl:Class 'Vegetable_Detail'
	'Vegetable_Detail' Class has an association relationship with 'Food_Display_Row' Class and is related by 'hasVegetable_Detail' ontology ObjectProperty. In triplet statement, we could say 'Food_Display_Row' -'hasVegetableDetail' -'Vegetable_Detail'.	owl:ObjectProperty 'hasVegetable_Detail' >>> rdfs:domain 'Food_Display_Row' >>> rdfs:range 'Vegetable_Detail'
	One instance in 'Food_Display_Row' must have exactly one individual of 'Vegetable_Detail'.	owl:Class 'Food_Display_Row' >>> owl:Restriction >>> owl:onProperty >>> owl:ObjectProperty 'hasVegetable_Detail', owl:minCardinality 1, owl:maxCardinality 1
Update		
<iAttrib>Vegetables <iAttrib>Vegetables_Total <iClass>Food_Display_Row <iType>xs:decimal <iMin>1<iMax>1	Attribute 'Vegetables' is renamed to 'Vegetables_Total'.	owl:DatatypeProperty 'dpVegetables'→ owl:DatatypeProperty 'dpVegetables_Total'
	'dpVegetable_Total' is the ontology DatatypeProperty of 'Food_Display_Row' Class, with decimal data type.	owl:DatatypeProperty 'dpVegetables_Total' >>> rdfs:domain 'Food_Display_Row' >>> rdfs:range decimal
	One instance in 'Food_Display_Row' must have exactly one value of 'Vegetable_Total'.	owl:Class 'Food_Display_Row' >>> owl:Restriction >>> owl:onProperty >>> owl:DatatypeProperty 'dpVegetables_Total', owl:minCardinality 1, owl:maxCardinality 1
Move		
<iAttrib>Orange_Vegetables <iClass>Food_Display_Row <iClass>Vegetable_Detail	Attribute 'Orange_Vegetables' is no longer describe 'Food_Display_Table', but now describe 'Vegetable_Detail'. Therefore 'dpOrange_Vegetables' is now the DatatypeProperty of 'Vegetable_Detail' Class.	~~owl:DatatypeProperty 'dpOrange_Vegetables' >>> rdfs:domain 'Food_Display_Row', owl:Class 'Food_Display_Row' >>> owl:Restriction >>> owl:onProperty >>> owl:DatatypeProperty 'dpOrange_Vegetables',~~ owl:DatatypeProperty 'dpOrange_Vegetables' >>> rdfs:domain 'Vegetable_Detail', owl:Class 'Vegetable_Detail' >>> owl:Restriction >>> owl:onProperty >>> owl:DatatypeProperty 'dpOrange_Vegetables'

5.1. Completeness and correctness

We design the *delta* script by taking *minimality* into consideration. While ensuring there are minimum non-redundant *delta* list items, it is important that all the required changes are propagated in terms of (a) the *completeness* of the components and (b) the *correctness* of the semantics.

Table 3. Ontology individual propagation case study.

Delta script	Derived knowledge	OWL repesentation
Delete		
<iD>13110120 <iAttrib>Oils<iVal>.00000	The attribute 'Oils' and its value in an instance with ID 13110120 is deleted.	<~~Food_Display_Row rdf:about='uri:Food_Display_Row_13110120'~~> <~~dpOils rdf:datatype='xs:decimal'~~>~~.00000~~ <~~/dpOils~~>
Insert		
<iD>13110120<iAttrib>lang <iClass>Display_Name <iVal>'en'	A new attribute 'lang' that describe a 'Display_Name' instance with ID 13110120 is inserted with a value of 'en',	<Display_Name rdf:about='uri:Display_Name_13110120'><dplang rdf:datatype='xs:NCName'>en</dplang>
Update		
<iD>13110120<iAttrib> Portion_Display_Name <iClass>Food_Display_Row <iVal>"cup" <iVal>"cup or scoop"	The value of attribute 'Portion_Display_Name' that describes a 'Food_Display_Row' instance with ID 13110120 is updated from 'cup' into 'cup or scoop'.	<Food_Display_Row rdf:about='uri:Food_Display_Row_13110120'> <~~dpPortion_Display_Name rdf:datatype='xs:NCName'~~>~~cup~~ <~~/dpPortion_Display_Name~~> <dpPortion_Display_Name rdf:datatype='xs:NCName'>cup or scoop </dpPortion_Display_Name>

Table 4. SPARQL evaluation of the case study.

No.	SPARQL query	Query result
1	SELECT ?item ?oil WHERE { ?item :dpOils ?oil}	item: Food_Display_Row_13110100 — oil: .00000
2	SELECT ?d ?aRel ?r WHERE { ?aRel a owl:ObjectProperty. ?aRel rdfs:domain ?d. ?aRel rdfs:range ?r }	d: Food_Display_Table — aRel: hasFood_Display_Row — r: Food_Display_Row d: Food_Display_Row — aRel: hasDisplay_Name — r: Display_Name d: Food_Display_Row — aRel: hasVegetable_Detail — r: Vegetable_Detail
3	SELECT ?item ?pdn ?lang ?name WHERE { ?item :hasDisplay_Name ?dn. ?item :dpPortion_Display_Name ? pdn FILTER (datatype(?pdn) = xsd:string). ?dn :dplang ?lang. ?dn :dpDisplay_Name ?name }	item: Food_Display_Row_13110120 — pdn: cup or scoop — lang: en — name: Ice cream, rich item: Food_Display_Row_13110100 — pdn: cup — lang: en — name: Ice cream, regular
4	SELECT ?item ?ov ?dv ?sv ?othv WHERE { ?item :dpVegetables_Total ?vt. ?item :hasVegetable_Detail ?vd. ?vd :dpOrange_Vegetables ?ov. ?vd :dpDrkgreen_Vegetables ?dv. ?vd :dpStarchy_Vegetables ?sv. ?vd :dpOther_Vegetables ?othv }	item: Food_Display_Row_13110120 — ov: .00000 — dv: .00000 — sv: .00000 — othv: .00000 item: Food_Display_Row_13110100 — ov: .00000 — dv: .00000 — sv: .00000 — othv: .00000

Component *Completeness.* We evaluate the *completeness* of the ontology model component by comparing the base ontology with the propagated ontology. We observe the deleted, inserted and updated Class(es)/Concept(s) and/or Property(s). The difference between both ontologies should correspond to the *delta* script, and the propagated ontology model component should suit the updated data source schema. Figure 4(a) shows the base ontology of the case study in the Protégé[6] ontology editor environment in OWL. The ObjectProperty name is prefixed by 'has' and 'dp' prefixes the DatatypeProperty name. The propagated ontology result from our application is depicted in Figure 4(b). Without mentioning the prefixes, in Figure 4(b), we can tell that there is no longer a 'Grains' property. In addition,

'Vegetable_Detail', 'Display_Name' and 'lang' properties have been inserted. The 'Vegetables' name property has become 'Vegetables_Total'. All changes come under the *delta* script and suit the updated data structure in the case study.

Semantic *Correctness*. We evaluate the *correctness* of ontology tuple, relation and constraint semantics in all operations (delete, insert, update and move). The propagated ontology should have the correct semantics in all of its tuple(s), relation(s) and constraint(s). In our evaluation, four SPARQL [25] queries are executed against the case study. These queries are detailed in Table 4. Query 1 results in there being only one instance having the 'Oils' value, since the 'Oils' attribute value from the instance with ID 13110120 has been deleted. Query 2 shows the association relationship between concepts in the propagated ontology. The result depicts that the relationship between the inserted concepts has been set in order, based on the *delta* and the updated data structure. Query 3 shows that all of the inserted attribute values of 'lang' and 'Display_Name' of all instances have been propagated. In addition, the list of results proves that the updated data type for 'Portion_Display_Name' has changed to string. Query 4 demonstrates that the 'Orange_Vegetables', 'Drkgreen_Vegetables', 'Starchy_Vegetables' and 'Other_Vegetables' have been moved and have become the attribute of 'Vegetable_Detail'.

The combined results of the updated ontology in Figure 4(b) and the SPARQL query results in Table 4 demonstrate that the *delta* script is *complete* and it contains accurate semantic information to update the base ontology. In other words, there is no information loss during the propagation process.

5.2. Performance

This section evaluates the data-driven dynamic ontology *performance*, especially in the propagation process. We evaluate two components of *performance*: (a) *operation cost*; and (b) *processing time*.

We use two data change scenarios to represent real world application trends. In this experiment, we make an assumption about the data class and attribute. The structural figures are adopted from the original Food Display Table[3] XML data. Predefined variables in representing schema and instances are used. In the schema part, n_c is the number of classes and n_a is the number of attributes with exactly one occurrence. In the data instance part, n_i is the total number of instance blocks in the data. Therefore, from the original Food Data Table XML data, we get $n_c = 2$ and $n_a = 26$.

For the operation cost, we compare the cost unit for propagating the ontology by means of Jena API against the cost unit for rebuilding the ontology from the current data, with the availability of current and previous data, schema and *delta* script. Especially for the operation cost evaluation, we define the unit of rebuild cost. We need two operations, scanning and mapping, to translate a single class or attribute into ontology form. In addition, one operation is needed to map each instance data into an ontology individual. The total operation cost for rebuilding the current ontology from current XML data is equal to $n_i(n_c + n_a) + 2n_c + 2n_a \rightarrow (n_i + 2)(n_c + n_a)$. The propagation operation cost is based on the number of methods that we call using Jena API, as seen in Table 5.

For the processing time, we compare the time elapsed for our propagation method against rebuilding the current ontology. In order to do so, we compare our web-based application using Jena API and *XML2OWL* [6] as a tool in rebuilding the populated ontology. *XML2OWL* uses XSLT (XML Stylesheet Language Transformation) to extract XSD from XML data, to create the OWL model from XSD and to translate XML data into ontology instances. For the processing time evaluation, we use the following hardware and software: Apple MacBook Pro (2.4 GHz Intel Core 2 Duo processor, 4 GB, 1067 MHz, DDR3 RAM), PHP 5.3.2, Apache 2.0.63 and Tomcat 6.0.26.

5.2.1. Scenario 1. In the first scenario (**Sc1**), the data instance is always added over time while the data structure remains the same. This normally occurs in cases such as our case study, or data relating to product or shop catalogues and membership information.

First, we need to obtain the operation cost for the propagation in **Sc1**. In order to do the propagation for **Sc1**, we need to recall all properties in the ontology and then insert each added individual value from the *delta* script. The cost of recalling is equal to the total number of all classes and attributes, which is $n_c + n_a$. The cost of inserting each individual value is $n_{i+}(n_c + 3n_a)$, where n_{i+} is the number of added instance blocks in the data and $(n_c + 3n_a)$ is the total Jena API methods participating in this propagation. The number of Jena API methods applied for each class in n_c is only 1, which creates an individual. On the other hand, every attribute in n_a needs three methods before it can be mapped into the ontology. These are creating a literal, creating a statement and adding the statement into the ontology model. The total operation cost in **Sc1** propagation is equal to $n_c + n_a + n_{i+}(n_c + 3n_a)$.

We consider that each insert operation adds 10 new instance blocks in the data; therefore the number of added instance blocks in the data $n_{i+} = 10$. The change occurs 10 times (from 10 to 100 total instances data). Figure 5(a) depicts a

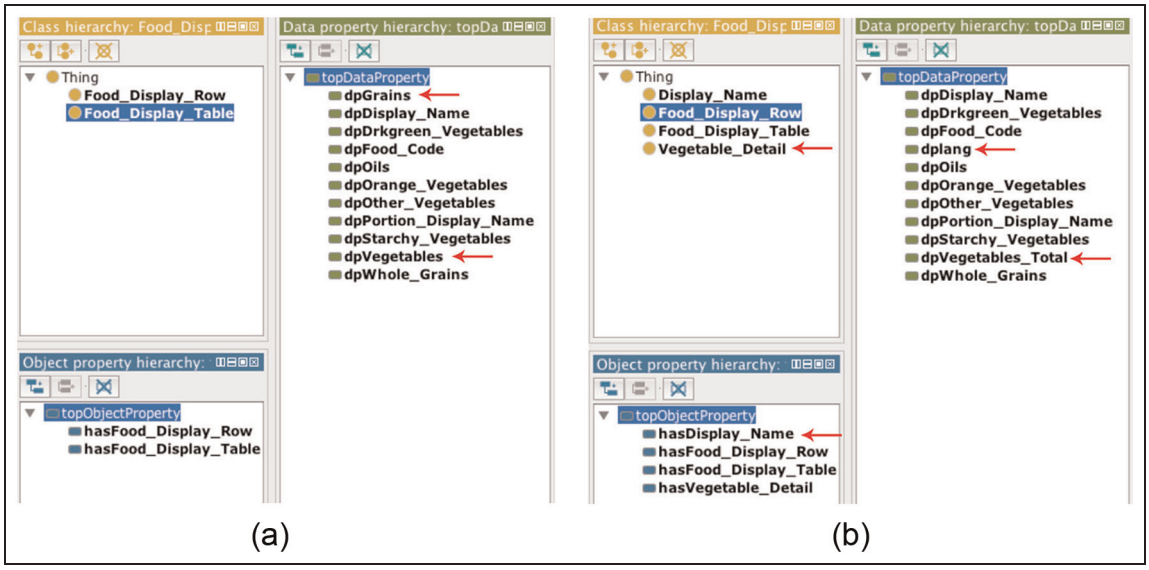

Figure 4. (a) Food Display Table base ontology. (b) Updated Food Display Table ontology.

Table 5. Jena API method for propagation.

No.	Operation	Jena method component
1	deleteCardinality	`listRestrictions(), remove()`
2	deleteStatement	`listStatements(), remove()`
3	deleteClass	`getOntClass(), remove()`
4	deleteAssosiationRelation	`getObjectProperty(), remove()`
5	deleteAttribute	`getDatatypeProperty(), remove()`
6	createCardinality	`createMinCardinalityRestriction() / createMaxCardinalityRestriction(), addSuperClass()`
7	createStatement	`createStatement() / createLiteralStatement(), add()`
8	createClass	`createClass()`
9	createAssosiationRelation	`createObjectProperty()`
10	createAttribute	`createDatatypeProperty()`
11	createIndividual	`createIndividual()`
12	setDomain	`setDomain()`
13	setRange	`setRange()`
14	renameComponent	`renameResource()`

comparison of the operation cost between the propagation and the rebuild method. Since we define $n_c = 2$, $n_a = 26$ at the beginning of Section 5.2 and the operation cost of rebuilding is equal to $(n_i + 2)(n_c + n_a)$, the cost of rebuilding is raised from 336 to 2856 operation cost units. The propagation method operation cost is steady at 828 operation cost units, since the delta always holds only 10 additions over time. The propagation cost is slightly higher in the first 25–30% of the whole changes. However, the propagation method gains efficiency in operation costs in most insertion cases of **Sc1**. In addition, it gives more efficiency in operation costs over time.

XML2OWL requires a great deal of resources to rebuild the collective amount of instances over time. This potentially creates a huge drawback in applications where the structural representation remains stable and yet data instances are regularly added or changed. In contrast, the propagation only needs to add 10 instances over time. A comparison of the total processing time can be seen in Figure 5(b), showing how the propagation method in **Sc1** gives results in an advantage in all processes by saving resources against rebuilding the whole ontology.

The evaluation result in real application processing time does not result in any drawback in the first 25–30% of the total changes. This happens because of the difference between the calculation and comparison methods. However, the results show that the propagation method is still superior to the rebuild method.

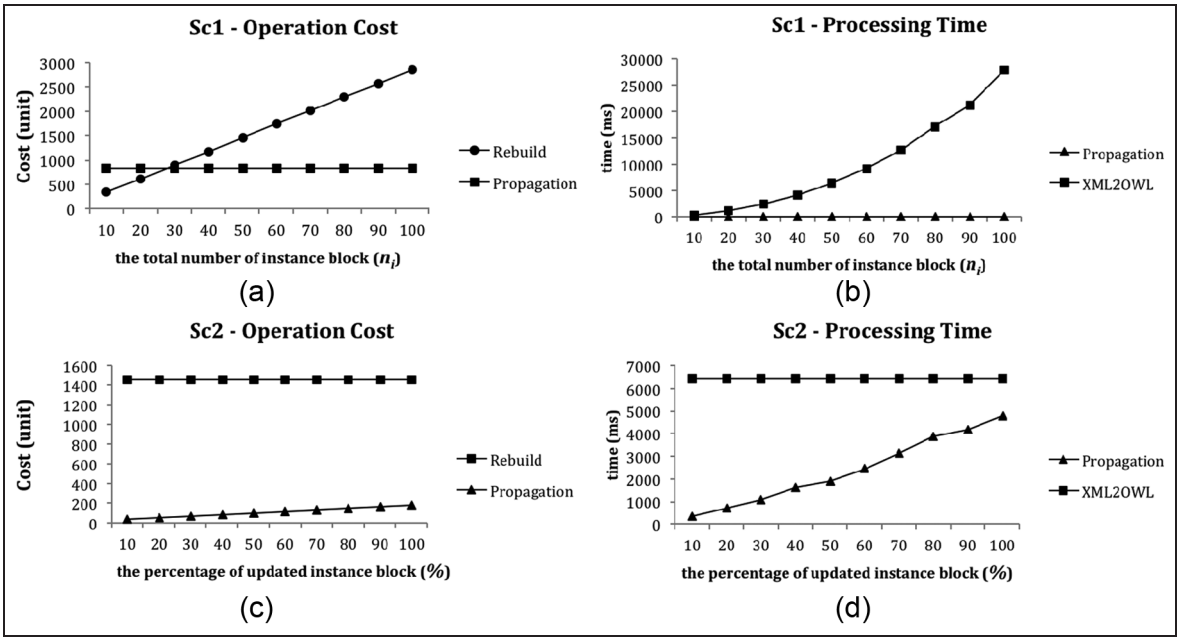

Figure 5. Performance evaluation, where the number of classes n_c = 2 and the number of attributes n_a = 26. (a) Operation cost comparison in **Sc1**. (b) Process time comparison in **Sc1**. (c) Operation cost comparison in **Sc2**. (d) Process time comparison in **Sc2**.

5.2.2. Scenario 2. In the second scenario (**Sc2**), the data instance is always updated over time, with no addition of the instance block, and the structure remains the same. This scenario might occur to application configuration data. For the experiment, we assume that the amount of instance block data is 50, therefore $n_i = 50$.

First, we need to obtain the operation cost for the propagation. In order to do the propagation for **Sc2**, we have to recall all properties in the ontology and perform the individual updates. The cost of recalling the properties is equal to $n_c + n_a$. The cost of updating individual values is $n_{i\wedge}(3n_a)$. $n_{i\wedge}$ is the number of updated instance block trees in the XML data and $3n_a$ is the total Jena API method participating in this propagation. These methods are for creating a literal, listing a statement and updating the statement. The total cost is equal to $n_c + n_a + n_{i\wedge}(3n_a)$.

Figure 5(c) depicts the operation cost comparison between the propagation and the rebuild methods in regard to the percentage of updated instances, where it increases from 10 to 100% by 10%. Since we define $n_c = 2$, $n_a = 26$ at the beginning of Section 5.2 and the operation cost of rebuilding is equal to $(n_i + 2)(n_c + n_a)$, the rebuild method cost is 1456 operation units. The propagation cost increases from 43 to 178 operation units, since there are increasing changes recorded in the *delta* script. However, the propagation method gives higher efficiency in operation costs.

The processing time evaluation in **Sc2** uses the same assumption of data and changes as in the cost evaluation, where the amount of updated instance blocks increases from 10 to 100% of total instances. Figure 5(d) shows the results of the processing time calculation evaluation. Even though in 100% of instance changes the propagation method is close to the rebuilding cost, the propagation method outperforms the rebuilding scenario in all cases.

6. Conclusion

Our research to develop a data-driven dynamic ontology model is motivated by the lack of a global standard and a common understanding of the community's knowledge repository. Our model comprises ontology base creation and ontology propagation. In the propagation process, we propose a novel *delta* script as a crucial tool in the propagation process. It enables remote ontology updates by only sending the minimum but complete updates.

As the proof of concept, we took XML data format representation in the case study and created the mapping of edit operations derived from the *delta* script list into a Jena API method. We evaluated the propagation captured by the *delta* script through various measures including *completeness, correctness* and *performance*. SPARQL is used to prove that the *delta* script created by our model holds minimum non-redundant data changes, yet it contains complete semantic information. Finally, we compared the operation cost and processing time of rebuilding the ontology against the propagation

method. The result indicates that the propagation model is lightweight in most cases. In future, we will apply this concept to a multiple source of knowledge where ontology merging is required to evaluate the applicability of our propagation in such an environment.

Funding

D.H. Fudholi (first author) is a recipient of La Trobe University Postgraduate Research Scholarship from La Trobe University, Australia.

Notes

1. http://www.thaiopensource.com/relaxng/trang.html
2. http://schemaspy.sourceforge.net
3. http://www.cnpp.usda.gov/Innovations/DataSource/MyFoodapediaData.zip
4. http://oxygenxml.com
5. http://jena.apache.org
6. http://protege.stanford.edu

References

[1] Calegari S and Ciucci D. Integrating fuzzy logic in ontologies. In: *Proceedings of the 8th international conference on enterprise information systems: Databases and information systems integration (ICEIS Part 2)*, 2006, pp. 66–73.

[2] Fortuna B, Grobelnik M and Mladenic D. OntoGen: Semi-automatic ontology editor. *Human Interface and the Management of Information. Interacting in Information Environments*. Lecture Notes in Computer Science, Vol. 4558. Berlin: Springer, 2007, pp. 309–318.

[3] Lee C, Kao Y, Kuo Y and Wang M. Automated ontology construction for unstructured text documents. *Data & Knowledge Engineering* 2007; 60: 547–566.

[4] Dahab MY, Hasan HA and Rafea A. TextOntoEx: Automatic ontology construction from natural English text. *Expert Systems with Applications* 2008; 34: 1474–1480.

[5] Ferdinand M, Zirpins C and Trastour D. Lifting XML Schema to OWL. In: *Proceedings of the 4th international conference on web engineering (ICWE)*, Munich, 2004, pp. 354– 358.

[6] Bohring H and Auer S. Mapping XML to OWL ontologies. In: *Proceedings of the 13th Leipziger Informatik-Tage*, Leipzig, 2005, pp. 147–156.

[7] García R. A Semantic Web approach to digital rights management. PhD thesis, Universitat Pompeu Fabra, Spain, 2006.

[8] Ghawi R and Cullot N. Building ontologies from XML data sources. In: *Proceedings of the 1st international workshop on modelling and visualization of XML and Semantic Web data (MoViX)*, Linz, 2009, pp. 480–484.

[9] Bedini I, Matheus CJ, Patel-Schneider PF, Boran A and Nguyen B. Transforming XML schema to OWL using patterns. In: *Proceedings of the 5th IEEE international conference on semantic computing (ICSC)*, Palo Alto, CA, 2011, pp. 102–109.

[10] Zhang L and Li J. Automatic generation of ontology based on database. *Journal of Computational Information Systems* 2011; 7(4): 1148–1154

[11] Zhou X, Xu G and Liu L. An approach for ontology construction based on relational database. *International Journal of Research and Reviews in Artificial Intelligence* 2011; 1(1): 16–19.

[12] Li M, Du X and Wang S. Learning ontology from relational database. In: *Proceedings of the fourth international conference on machine learning and cybernetics*, Guangzhou, 2005; Vol. 6: 3410–3415.

[13] Zablith F, Antoniou G, d'Aquin M et al. Ontology evolution: a process-centric survey. In: *The knowledge engineering review*. Cambridge: Cambridge University Press, 2013.

[14] Stojanofic L. Methods and tools for ontology evolution. PhD thesis, FZI-Research Center for Information Technologies at the University of Karslruhe, 2004.

[15] Sari AK, Rahayu W and Bhatt M. An approach for sub-ontology evolution in a distributed health care enterprise. *Information Systems Journal* 2013; 38(5): 727–744.

[16] Zablith F. Evolva: A comprehensive approach to ontology evolution. In: *Proceedings of the PhD symposium of the 6th European Semantic Web conference (ESWC-09)*, Heraklion, Greece, 2009, pp. 944–948.

[17] Guerrini G, Mesiti M and Rossi D. Impact of XML schema evolution on valid documents. In: *Proceedings of the 7th annual ACM international workshop on web information and data management (WIDM)*, Bremen, Germany, 2005, pp. 39–44.

[18] Selkow SM. The tree-to-tree editing problem. *Information Processing Letters* 1977; 7(4): 184–186.

[19] Cobéna G, Abiteboul S and Marian A. Detecting changes in XML documents. In: *Proceedings of the 18th international conference of data engineering (ICDE)*, San Jose, CA, 2002, pp. 41–52.

[20] Wang Y, DeWitt DJ and Cai JY. *X-Diff*: An effective change detection algorithm for XML documents. In: *Proceedings of the 19th international conference on data engineering (ICDE)*, Bangalore, 2003, pp. 519–530.

[21] Leonardi E, Bhowmick SS and Madria S. Xandy: Detecting changes on large unordered XML documents using relational databases. In: *Proceedings of the 10th international conference on database systems for advanced applications (DASFAA)*, Beijing, 2005, pp. 711–723.
[22] Baqasah A, Pardede E, Rahayu W and Holubova I. On change detection of XML schemas. In: *Proceedings of the 11th IEEE international symposium on parallel and distributed processing with applications (ISPA)*, Melbourne, 2013: 974–982.
[23] Cobéna G, Abdessalem T and Hinnach Y. A comparative study of XML diff tools, http://www.deltaxml.com/support/documents/articles-and-papers/is2004.pdf (2004, accessed June 2013).
[24] Dean M and Schreiber G (eds). OWL web ontology language reference, http://www.w3.org/TR/owl-ref/ (2004, accessed March 2014).
[25] Prud'hommeaux E and Seaborne A (eds). SPARQL query language for RDF, http://www.w3.org/TR/rdf-sparql-query/ (2008, accessed March 2014).