Journal of The Korea Society of Computer and Information
Vol. 21 No. 10, pp. 1-10, October 2016
www.ksci.re.kr
http://dx.doi.org/10.9708/jksci.2016.21.10.001

Automatic Generation of Machine Readable Context Annotations for SPARQL Results

Ji-Woong Choi *

Abstract

In this paper, we propose an approach to generate machine readable context annotations for SPARQL Results. According to W3C Recommendations, the retrieved data from RDF or OWL data sources are represented in tabular form, in which each cell's data is described by only type and value. The simple query result form is generally useful, but it is not sufficient to explain the semantics of the data in query results. To explain the meaning of the data, appropriate annotations must be added to the query results. In this paper, we generate the annotations from the basic graph patterns in user's queries. We could also manipulate the original queries to complete the annotations. The generated annotations are represented using the RDFa syntax in our study. The RDFa expressions in HTML are machine-understandable. We believe that our work will improve the trustworthiness of query results and contribute to distribute the data to meet the vision of the Semantic Web.

▸ Keyword : Linked Open Data, SPARQL, RDFa, Provenance, Semantic Web

I. Introduction

W3C 주도로 진행되고 있는 LOD(Linked Open Data) 프로젝트의 확산으로 인해 시맨틱 웹 상에서 공개적으로 접근 가능한 다양한 분야의 데이터 셋이 빠르게 증가하고 있으며 이들 데이터를 근간으로 한 활용 사례 또한 꾸준히 보고되고 있다[1,2].

공개 데이터의 취득 수단은 RDF를 위한 질의 언어인 SPARQL이다[3,4]. W3C의 표준문서 [5-7]에 따르면, SPARQL 결과 데이터는 JSON, CSV, TSV, XML 포맷 중 하나로 표현될 수 있다. 이들 네 가지 포맷은 각기 고유의 신택스를 갖지만 공통적으로 모두 테이블 모델을 표현한다. SPARQL 결과를 표현하는 테이블 모델은 각각의 셀 데이터를 값과 타입으로만 설명한다. 타입은 구체적으로 URI 혹은 리터럴이 될 수 있다. RDF 데이터 모델의 구성단위인 트리플이 URI와 리터럴로 이루어져 있기 때문이다.

표준 SPARQL 질의 결과 형식은 앞서 언급한 대로 질의 결과의 의미(semantics)에 대한 설명이 누락되어 있다. 달리 말하면, RDF 모델은 데이터의 의미를 다른 데이터와의 관계 정의에 의해 표현한다. 그러나 표준 SPARQL 질의 결과 형식으로 추출된 데이터는 다른 데이터와의 관계가 단절되어 있다. 즉, 질의 결과에 포함된 데이터는 문맥이 결여되어 있다.

공개 데이터 소스에서 추출한 데이터를 가공하여 다시 웹으로 유통시키는 응용은 일반적이다. 이러한 응용들 중에서 그들이 유통시킬 데이터의 문맥을 부가함으로써 그 데이터를 사용할 다른 잠재적인 응용에 대해 그 데이터의 재사용성과 신뢰성을 향상시키고자 하는 응용의 경우 기존의 SPARQL 질의 결과 형식으로는 데이터에 대한 문맥의 획득이 어렵기 때문에 추가적이 노력이 소요된다[8].

본 논문에서는 이러한 문제를 해결하기 위하여 SPARQL 질의 결과에 포함된 데이터의 의미를 설명하는 문맥 정보를 자동적으로 추가하여 HTML 문서를 생성하는 방법을 제안한다. 본 논문에서 생성하는 HTML 문서를 보다 정확히 기술하면 HTML+RDFa(Resource Description Framework in Attributes) 문서이다. HTML+RDFa 문서는 HTML 문서이기 때문이다[9]. 본 논문에서 생성하는 HTML 문서 내의 RDFa 요소는

• First Author: Ji-Woong Choi, Corresponding Author: Ji-Woong Choi
*Ji-Woong Choi (iamjwchoi@gmail.com), School of Computer Science and Engineering, Soongsil University
• Received: 2016. 08. 26, Revised: 2016. 08. 29, Accepted: 2016. 09. 08.

문맥 정보를 표현한다. RDFa는 HTML 문서 내에서 RDF 모델을 표현하기 위한 기술이다[9]. RDFa 요소들은 웹 브라우저에 의해서 인간을 위해 렌더링 되지는 않지만 기계에 의해 의미가 읽혀질 수 있다. RDFa로 표현될 문맥 정보의 근거와 범위는 사용자의 SPARQL 질의문이 포함하고 있는 BGP(Basic Graph Pattern)이다. SPARQL 결과는 기본적으로 질의 대상 RDF 모델에서 BGP를 만족하는 튜플들로 구성되어 있기 때문이다. 따라서, 본 논문에서는 SPARQL 질의에서 추출한 BGP를 문맥 정보 생성을 위한 템플릿으로서 사용하며 SPARQL 질의 결과에 포함된 개별 데이터들을 템플릿에 채워 넣어 문맥 정보를 완성한 후 RDFa로 출력한다.

본 연구의 결과는 SPARQL 질의 결과 데이터를 기반으로 동적 웹 페이지를 생성하는 응용에서 웹 페이지에 질의 결과 데이터뿐만 아니라 그 데이터를 설명하는 의미 정보를 함께 포함시키고자 할 때 적용될 수 있다.

본 논문의 구성은 다음과 같다. 2장에서는 관련 연구와 배경 지식을 기술한다. 3장에서는 SPARQL 질의 결과에 포함된 데이터의 의미를 설명하는 문맥 정보를 자동적으로 추가하여 HTML 문서를 생성하는 과정과 방법을 상세히 기술한다. 4장에서는 3장의 내용을 구현한 시스템을 사용해 평가를 수행하며, 5장에서는 결론을 맺는다.

II. Related Works

본 장은 관련 연구로서 시맨틱 웹 데이터에 대한 출처(provenance) 정보의 생성과 교환에 관한 연구들과 HTML 문서 내에 시맨틱 정보를 표현하기 위하여 고안된 RDFa를 활용하는 연구들을 소개하고 분석한다.

1. Provenance on the Semantic Web Data

데이터 출처는 연구 [10]에 의하면 데이터에 대한 “origin”, “lineage”, “source”로 정의된다. 이 개념은 폭넓게 사용될 수 있으므로 연구 [11]에서 제안한 데이터 출처를 보다 세분화하여 설명할 수 있는 “what”, “when”, “where”, “how”, “who”, “which”, “why”의 7가지 관점들이 이후 여러 연구에서 “why-provenance”와 같은 방식으로 특정되어 차용되고 있다.

2009년 연구 [12]이 주장한 시맨틱 웹 데이터의 출처 제공에 관한 필요성은 이후 W3C에 의해 표준화 활동으로 수렴되었으며 2013년 W3C는 시맨틱 웹 데이터의 출처에 관한 표준 문서 셋 [13]을 발표하였다. 이 표준 문서 셋은 PROV로 명명된 데이터 출처 정보를 제공하기 위한 데이터 저장소 측이 준수해야 할 데이터 모델과 출처 정보 표현을 위한 표준 어휘(vocabulary)들의 정의로 구성된다. PROV 모델은 ‘who’, ‘what’, ‘when’, ‘how’의 4가지 출처 정보의 제공에 초점을 맞추어 설계되었다[14].

연구 [1]에 의하면, LOD 기반 데이터 셋에 PROV 모델과 어휘가 적용된 비중은 미미하다. 이 결과는 연구 [1]의 조사 시점이 PROV 표준의 발표 시점과 가까운 것에 기인한다. 그러나 이 연구에서 주목해야 할 점은 출처 정보를 표현하기 위한 어휘로 간주될 수 있는 DCMI Metadata Terms [15]와 같은 W3C 비표준 어휘들이 이미 높은 비중으로 많이 사용되고 있다는 것이다. 이러한 점을 고려하여 본 연구에서는 RDFa로 표현할 데이터에 대한 메타데이터를 W3C 표준 어휘만으로 한정하지 않음으로 문맥 정보라는 표현을 사용하였다.

데이터의 원천인 데이터 저장소 측에서 출처 정보를 제공하도록 하는 W3C의 표준화 방향과는 별도로 연구 [16-18]은 SPARQL 질의 결과에 대한 출처 정보를 SPARQL 질의문 분석을 통해 도출하고자 한다. 질의 결과에 대한 출처 정보란 질의 결과의 생성에 영향을 미친 질의문 구문요소의 규명(why-provenance) 혹은 연산 순서의 규명(how-provenance)이다. 이러한 정보를 획득하기 위하여 이들 연구에서는 SPARQL 질의문을 SPARQL 대수식[4]으로 변형하여 분석한다. 질의 결과에 대한 설명은 질의 결과 내의 하나의 튜플(tuple) 단위로 제공된다. [16]와 [17]는 how-provenance에 관한 연구이며 [18]은 why-provenance에 관한 연구이다.

연구 [16-18]은 그들 자신의 고유의 방법으로 생성해 낸 출처 정보를 시맨틱 웹에서 유통될 수 있는 포맷으로 표현하는 것에 대한 고려는 없다. 그러나 이러한 정보를 표현할 수 있는 표준화된 RDF 어휘도 개발되어있지 않다. 따라서 이러한 어휘를 본 연구에서 자체 개발하여 RDFa로 표현한다 하더라도 현재로서는 그 RDFa 표현의 범용성을 확보하기 어렵다.

본 연구에서 SPARQL 질의문으로부터 추출하는 요소는 질의 결과 튜플에 영향을 미친 BGP들 뿐이다. 즉, SPARQL 질의문의 다른 구문 요소는 추출 대상이 아니다.

2. HTML+RDFa Authoring

RDFa는 문서의 웹과 데이터의 웹 사이의 간극을 연결하기 위하여 고안된 기술이다. HTML 문서 내에 삽입된 RDFa 표현은 콘텐트의 의미를 기계가 이해할 수 있도록 만들기 때문에 그 문서의 가치를 증대시킬 수 있음은 자명하다. 그러나 수작업에 의한 RDFa 주석(annotation) 첨가는 효율성이 떨어지므로 이에 대한 자동화 내지 사용자 편의성 제고를 위한 연구들이 있어 왔다.

연구 [19]은 의미 주석이 없는 웹 문서를 입력으로 하여 RDFa 주석이 자동으로 첨가된 웹 문서를 출력해주는 웹 서비스이다. 이 연구의 접근법은 자동화된 RDFa 주석 첨가가 가능하지만 웹 페이지 저작자가 자신의 웹페이지에 RDFa가 추가되는 과정에 개입할 여지가 없기 때문에 그 결과에 따른 만족도에 따라 활용성이 결정된다는 한계를 가지고 있다. 그러나 본 연구에서 데이터의 의미 주석을 위한 템플릿으로서 채택한 질의문 내의 BGP는 질의문 생성자가 LOD 데이터 셋으로부터 얻기 원하는 데이터의 의미를 한정할 목적으로 작성한 것이기 때문에 그 의도를 그대로 반영한다.

연구 [20]은 일종의 RDFa 주석 작업을 위한 편집기로서 사용자에게 RDFa 주석 첨가 혹은 첨가된 주석의 의미파악을 위한 직관적인 환경을 제공한다. 그러나 이것은 편집기이기 때문에 프로그램에 의해 동적으로 생성되는 웹 페이지를 위해서는 사용될 수 없다.

III. The Proposed Approach

1. Turning a Query to an Algebra Expression

문맥 정보를 포함한 SPARQL 질의 결과를 생성하기 위한 첫 번째 작업은 사용자의 원본 질의문을 SPARQL 대수식으로 변환하는 것이다. 그 이유는 후술할 질의문 조작과 트리플 추출 작업을 용이하게 수행하기 위함이다. 그림 1을 통해 SPARQL 대수식 기반의 작업이 SPARQL 질의문 기반의 작업보다 용이한 이유를 설명하고자 한다. 그림 1의 상단이 SPARQL 질의문이며 하단이 SPARQL 대수식이다. 그림 1의 두 표현은 동등한 의미를 갖는다.

```
01 PREFIX  ex:    <http://ex.com/>
02
03 SELECT  ?a ?c ?e ?i
04 WHERE
05   {   { ?a  ex:b  ?c ;
06           ex:d  ?e
07         OPTIONAL
08           { ?c  ex:f  ?g }
09       }
10     UNION
11       { ?a  ex:d  ?c }
12     ?a  ex:h  ?i
13   }
```

```
01 (project (?a ?c ?e ?i)
02   (sequence
03     (union
04       (conditional
05         (bgp
06           (triple ?a <http://ex.com/b> ?c)
07           (triple ?a <http://ex.com/d> ?e)
08         )
09         (bgp (triple ?c <http://ex.com/f> ?g)))
10       (bgp (triple ?a <http://ex.com/d> ?c)))
11     (bgp (triple ?a <http://ex.com/h> ?i))))
```

Fig. 1. A SPARQL and its SPARQL algebra expression

SPARQL 질의문에서는 복수개의 트리플을 축약된 형태로 표현하는 것이 허용된다. 그러나 SPARQL 대수식은 SPARQL 질의문의 축약형 트리플들을 독립적인 트리플들로 분리시킨 형태로 표현한다. 그림 1 상단의 5~6행은 두 개의 트리플을 축약된 형태로 표현한 반면 그림 1 하단의 6~7행은 이를 분리시켜 표현하고 있다. SPARQL 대수식의 이러한 특성은 트리플 추출 작업 시 편리함을 제공한다.

SPARQL 대수식은 원본 SPARQL 질의문을 처리하기 위한 연산의 순서 그리고 개별 연산의 범위를 구조화하여 표현하며 식(expression) 표현 방식에 있어서 prefix 노테이션 방식을 취한다. 따라서 SPARQL 대수식은 왼쪽에서 오른쪽으로의 단방향 탐색만으로 질의문 조작 작업을 위한 구문요소들의 자격 파악을 가능케 한다. 그림 1 상단에서는 10행의 UNION 연산자가 발견된 후에야 5~9행이 UNION 연산자의 피연산자임을 파악할 수 있다. 반면에 그림 1 하단에서는 UNION 연산자가 3행에서 발견되므로 4~10행의 자격이 UNION 연산자의 피연산자임을 미리 예측할 수 있다. 즉, SPARQL 대수식은 기계를 통한 단방향 구문 탐색의 구현을 용이하게 한다.

본 연구에서는 Apache Jena[21]의 ARQ API를 사용하여 사용자의 SPARQL 질의문을 컴파일한 결과로서 SPARQL 대수식을 획득한다.

2. Modification of SPARQL Algebra Expression

2.1 Building Tree from SPARQL Algebra Expression

본 연구에서는 SPARQL 대수식 기반의 작업이 용이하도록 SPARQL 대수식을 그래프로 변형한 후 사용한다. 그림 2는 그림 1의 SPARQL 대수식으로부터 유도한 트리이다.

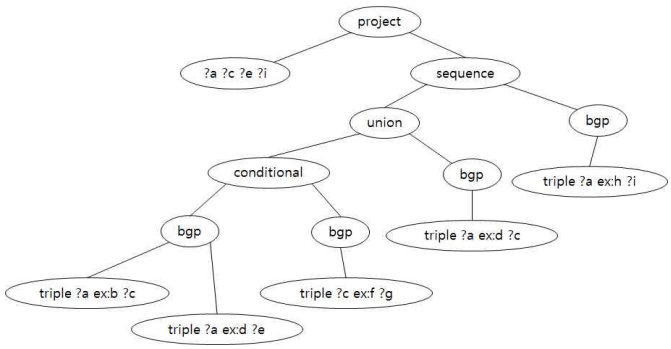

Fig. 2. The tree built from the SPARQL algebra expression in Fig. 1.

```
01 Node buildAlgebraTree(char[] A)
02 input: a SPARQL algebra expression A
03 output: the root node root of the tree derived from A
04
05 Node root ← null, current ← null, new ← null
06 char length ← A's length
07 char index ← 0
08 while (index < length) {
09   if (A[index] == '(') {
10     new ← newly created node
11     if (root == null)
12       Assign new to root and current.
13     else {
14       Add new to current as a child node.
15       current ← new
16     }
17   } else if (A[index] == ')') {
18     current ← current's parent node.
19   } else
20     Append A[index] to current's label.
21 }
22 return root
```

Fig. 3. Algorithm to build tree from SPARQL algebra expression

그림 3은 SPARQL 대수식을 그래프로 변형하는 과정을 설명한다. 그림 3의 알고리즘은 SPARQL 대수식에서 문자 '('가 출현할 때마다 현재 노드(*current*)의 새로운 자식 노드(*new*)를 생성한 후 새로 생성된 노드를 현재 노드로 세팅한다. 그리고 ')' 문자가 출현할 때마다 현재 노드의 부모 노드를 현재 노드로 세팅한다. 그 이외의 SPARQL 대수식의 문자는 현재 노드의 레이블이 된다.

2.2 Ambiguity under UNION Operator

그림 4를 통해 UNION 연산자를 포함한 질의문의 결과를 설명하고자 할 때 트리플 선택 문제에 있어서 발생하는 모호성을 설명하고자 한다.

Query	**SELECT** ?name **WHERE** { ?x a foaf:Person #① { ?x foaf:name ?name } #② **UNION** { ?x vCard:FN ?name } #③ }
Query Result	**name**
	"Alice"
	"Bob"

Fig. 4. Query with ambiguity

그림 4의 질의 결과인 "Alice"와 "Bob"은 변수 name에 바인딩 된 값들이다. 이 값들을 설명할 수 있는 트리플들의 집합은 경우에 따라 {①, ②}, {①, ③}, {①, ②, ③}이 가능하다. 세 집합 중 하나를 선택하여 질의 결과를 설명해야 하나 질의문과 질의 결과만으로는 판단이 불가능하다. 따라서 본 논문에서는 이러한 문제를 해결하기 위하여 그림 4의 질의문(사용자의 원본 질의문)을 SPARQL 대수식 수준에서 조작하여 그림 5와 같은 형태의 질의문으로 변형하고자 한다.

Query	**SELECT** ?name_1 ?name_2
	WHERE
	{
	{ ?x_1 a foaf:Person . #①
	?x_1 foaf:name ?name_1 } #②
	UNION
	{ ?x_2 a foaf:Person . #③
	?x_2 vCard:FN ?name_2 } #④
	}	

Query Result	**name_1**	**name_2**
	"Alice"	
		"Bob"

Fig. 5. Query without ambiguity

질의문 변형의 조건은 원본 질의문에 UNION 연산자가 1개 이상 존재하는 경우이다. 변형된 질의문은 그림 5와 같이 WHERE 절 내부에 UNION 연산자와 UNION 연산자의 피연자로만 구성된다. 하나의 UNION 피연산자 내부의 변수명들은 공통된 postfix로 끝나도록 수정되며 그 postfix는 다른 UNION 피연산자에서 사용된 postfix와 같지 않다. 마지막으로 SELECT 절의 변수명은 수정된 변수명 리스트로 치환된다. 이러한 과정을 거쳐서 수정된 질의문과 그 결과는 그림 4와는 달리 트리플 집합 {①, ②}를 사용하여 "Alice"를 설명할 수 있으며 트리플 집합 {③, ④}를 사용하여 "Bob"을 설명할 수 있다.

2.3 Transformation of Query with UNION Operators

그림 6은 SPARQL 대수식 트리에 union 노드가 존재할 경우 이를 제거한 복수개의 SPARQL 대수식 트리를 얻기 위한 알고리즘이다. union 노드가 제거된 복수개의 트리는 *treeList*로 명명된 리스트에 저장된다. 그림 7은 그림 2의 SPARQL 대수식 트리를 그림 6의 알고리즘에 적용한 결과 *treeList*에 저장된 두 개의 SPARQL 대수식 트리이다.

```
01 List seperateIntoTreesWithoutUNION(Node root)
02 input: the root node root of a SPARQL algebra tree
03 output: treeList, list of trees without UNION
04 List treeList // initialized to empty
05 Queue tempQ // initialized to empty
06 tempQ.enqueue(root)
07 while (tempQ is not empty) {
08  Node root' ← tempQ.dequeue()
09  removeUNION(root', treeList, tempQ)
10 }
11 return treeList
```

Fig. 6. Algorithm to build trees without "union" node

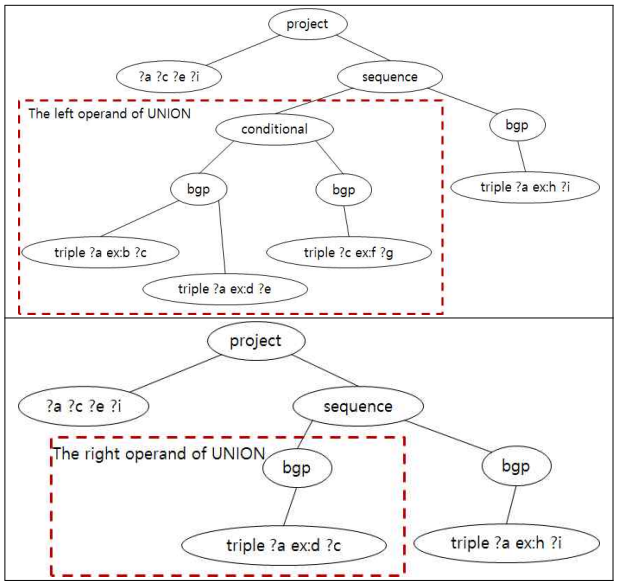

Fig. 7. The two tress derived from Fig. 2 by the algorithm in Fig. 6.

그림 6의 9행에서 호출되는 그림 8의 알고리즘은 1회 호출당 입력 트리에서 하나의 union 노드를 제거한다. 그림 8의 알고리즘은 입력 트리를 너비우선방식으로 탐색하며 레이블이 "union"인 노드를 찾는다. union 노드(*children*[*i*])가 발견되면 union 노드의 왼쪽 자식 노드(*lOperand*)를 union 노드의 부모 노드(*current*)의 *i*번째 자식 노드로 변경한다. 즉, 변경 전

*current*의 *i*번째 자식 노드는 union 노드이나 변경 후 *current*의 *i*번째 자식 노드는 *lOperand*이다. 그림 8의 13행은 변경된 트리의 사본을 *tempQ*에 삽입시킨다. 그림 6과 8에서의 *tempQ*는 union 노드가 존재하지 않는다고 결정되지 않은 트리들이 존재하는 곳이다. *tempQ*에 삽입된 트리들은 그림 6의 8~9행에 의해서 union 노드의 검출과 제거 작업을 거치게 된다. 그림 8의 14~18행은 union 노드의 오른쪽 자식 노드(*rOperand*)를 *current*의 *i*번째 자식 노드로 변경한 후 트리의 사본을 *tempQ*에 삽입시키고 프로시저의 동작을 종료한다. 그림 8의 23행은 트리의 모든 노드를 탐색한 결과 union 노드가 존재하지 않을 때 실행될 수 있으며 이러한 조건을 만족하는 트리는 *treeList*에 삽입된다.

```
01 void removeUNION(Node root, List treeList,
02                                  Queue tempQ)
03 Queue Q // initialized to empty.
04 Q.enqueue(root)
05 while (!Q.isEmpty()) {
06   Node current ← Q.dequeue()
07   Node[] children ← current.getChildren()
08   for (i = 0; i < the number of children; i++) {
09     if (label of children[i] == "union") {
10       // for left operand of UNION: 11~13 lines
11       Node lOperand ← the left child node of children[i]
12       children[i] ← lOperand
13       tempQ.enqueue(clone(root))
14       // for right operand of UNION: 15~17 lines
15       Node rOperand ← the right child node of children[i]
16       children[i] ← rOperand
17       tempQ.enqueue(clone(root))
18       return
19     } else
20       Q.enqueue(children[i])
21   } // end for
22 } // end while
23 treeList.add(root)
```

Fig. 8. Algorithm to remove a "union" node

그림 6과 8의 알고리즘은 질의문에 포함된 UNION 연산자의 개수와 위치에 관계없이 동작하도록 설계되었다. 예를 들면, 임의의 UNION 연산자의 피연산자에 UNION 연산자가 다시 출현하는 중첩된 형태도 처리할 수 있다. 그림 6과 8의 알고리즘은 사용자의 원본 질의문에 n개의 UNION 연산자가 존재할 경우 2^n개의 UNION 연산자가 제거된 SPARQL 대수식 트리를 생성하며 이들은 *treeList*에 저장된다.

UNION 연산자 제거가 완료되면 *treeList*에 저장된 각각의 트리에 고유의 postfix를 부여한다. 부여된 postfix는 해당 트리에 출현하는 모든 변수명 끝에 덧붙여진다. 이 과정에서 SPARQL 구문을 기준으로 설명하면 SELECT 절에는 출현하나 WHERE 절에는 출현하지 않는 변수명이 존재하면 SELECT 절의 변수목록에서 제거한다. 그림 7 하단 트리의 경우 이 트리에 부여된 postfix가 "_2"라고 가정하면 SELECT 절의 변수목록은 ?a ?c ?e ?i에서 ?a_2 ?c_2 ?i_2로 변경될 것이다.

변수명 수정이 완료되면 SPARQL 대수식 변형을 위한 마지막 절차인 *treeList*에 저장된 각각의 트리를 하나의 SPARQL 대수식 트리로 병합하는 작업을 수행한다. 원본 트리와 *treeList*의 트리들은 루트 노드에서부터 project 노드까지는 모두 동일하다. 따라서 병합 트리 또한 루트 노드에서부터 project 노드까지는 원본 트리와 동일하도록 만든다. project 노드의 왼쪽 자식 노드는 SPARQL 구문의 SELECT 절에 대응하며 오른쪽 자식 노드와 그 자식 노드들은 SPARQL 구문의 WHERE 절 이후에 대응한다. 따라서 병합 트리의 project 노드의 왼쪽 자식 노드의 레이블은 *treeList*의 트리들의 SELECT 절들의 변수 목록의 합이 되도록 만든다. 병합 트리의 오른쪽 자식 트리의 구조는 *treeList*의 각각의 트리에서 project 노드의 오른쪽 자식 트리들을 떼어내어 이들을 UNION 연산자의 피연산자로 취한 형태가 되도록 만든다. 이 때, *treeList*에 n개의 트리가 존재한다면 이들의 결합을 위해서는 병합 트리에 n-1개의 union 노드가 새로 추가된다. 그림 9는 그림 7의 두 SPARQL 대수식 트리를 하나로 병합한 결과이다. 그림 9의 union 노드의 왼쪽 자식 트리는 그림 7 상단의 project 노드의 오른쪽 자식 트리에서 유도되었으며 그림 9의 union 노드의 오른쪽 자식 트리는 그림 7 하단의 project 노드의 오른쪽 자식 트리에서 유도되었다.

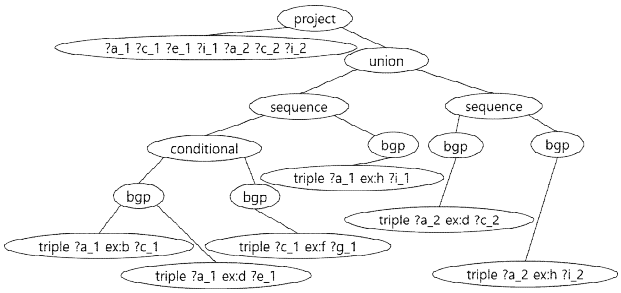

Fig. 9. A merged SPARQL algebra expression tree

그림 10은 4개의 요소를 갖는 *treeList*의 대수식 트리들을 병합한 결과를 단순화시켜 표현한 것이다. 그림 10의 A, B, C, D는 *treeList*에 있는 각각의 대수식 트리의 project 노드의 오른쪽 자식 트리이다. 그림 10 대수식 트리와 동등한 SPARQL 질의문의 WHERE 절은 {A'} UNION {B'} UNION {C'} UNION {D'}이다. 여기에서 A', B', C', D'는 각각 A, B, C, D와 동등한 SPARQL 구문이다.

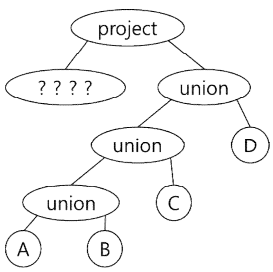

Fig. 10. Example tree with multiple "union" node

대수식 트리들의 병합이 완료되면 병합된 대수식 트리를

SPARQL 대수식으로 환원시킨다. 본 연구에서는 환원된 대수식을 Apache Jena[21]의 ARQ API를 사용하여 동등한 의미를 갖는 SPARQL 질의문으로 다시 변환시킨 후 이를 데이터 저장소에 질의한다.

3. Extraction of Basic Graph Patterns

본 연구에서는 *treeList*에 있는 각각의 대수식 트리로부터 BGP들을 추출하여 BGP 트리를 생성한다. 즉, *treeList*에 n개의 대수식 트리가 있을 경우 n개의 BGP 트리가 생성된다. BGP 트리들은 문맥 정보 생성을 위한 템플릿으로서 사용된다.

그림 11과 12는 복수개의 BGP 트리를 생성할 수 있는 알고리즘이다. 그림 11의 반복문 1회 실행 당 1개의 BGP 트리가 생성된다. 그림 11의 알고리즘에서는 각각의 BGP 트리의 루트 노드(*BGPRoot*)만을 생성한다. 그림 12는 하나의 대수식 트리를 깊이우선방식으로 탐색하며 트리플 수집과 *BGPRoot*의 자식 트리 생성을 병행한다. 그림 11과 12에서 사용되고 있는 변수 *current*는 BGPNode 타입으로서 전역 변수이다. 수집된 트리플은 BGP 트리에 존재하는 여러 개의 노드들 중 *current*가 가리키는 BGPNode에 저장된다. 하나의 BGPNode에는 복수개의 트리플을 저장할 수 있는 리스트가 존재한다.

```
01  List extractBGP(List treeList)
02  input: treeList, list of trees without UNION
03  output: BGPTreeList, list of trees of BGPs
04  List BGPTreeList // initialized to empty.
05  for (i = 0; i < the element count of treeList; i++) {
06   BGPNode BGPRoot ← newly created node
07   current ← BGPRoot
08   DFS(root node of treeList[i])
09   BGPTreeList.add(BGPRoot)
10  }
11  return BGPTreeList
```

Fig. 11. Algorithm to extract BGP from treeList

```
01  void DFS(Node v)
02  input: a node v in a tree in the treeList
03  // visiting v
04  if (Does the label of v start with "triple") {
05   Triple aTriple ← extract a triple from v
06   Append aTriple to the triple list in current.
07  }
08  // before visiting children of v
09  for (each child u of v) {
10   if (label of v == "conditional" &&
11                              Is u the left child of v) {
12    BGPNode new ← newly created node
13    Add new to current as a child node.
14    current ← new
15   }
16   DFS(u) // visiting u
17  }
18  // after visiting all children of v
19  if (label of v == "conditional")
20   current ← current's parent node
```

Fig. 12. Depth fisrt search algorithm to build a BGP tree

그림 12의 알고리즘은 대수식 트리의 방문한 노드(*v*)의 레이블이 "triple"로 시작하면 그 노드의 트리플만을 추출하여 *current*가 가리키는 BGP 트리 노드의 리스트에 추가한다. 노드 *v*의 레이블이 "conditional"이고 노드 *v*의 왼쪽 자식 노드를 방문할 차례가 되면 새로운 BGP 트리 노드를 생성하고 그 노드를 *current*의 자식 노드로 추가한다. 그리고 그 새 노드를 *current*가 가리키게 세팅한 후 *v*의 왼쪽 자식 노드를 방문한다. 노드의 레이블이 "conditional"인 노드(*v*)의 모든 자식 노드의 방문이 끝나면 *current*의 부모 노드를 새로운 *current*로 세팅한다.

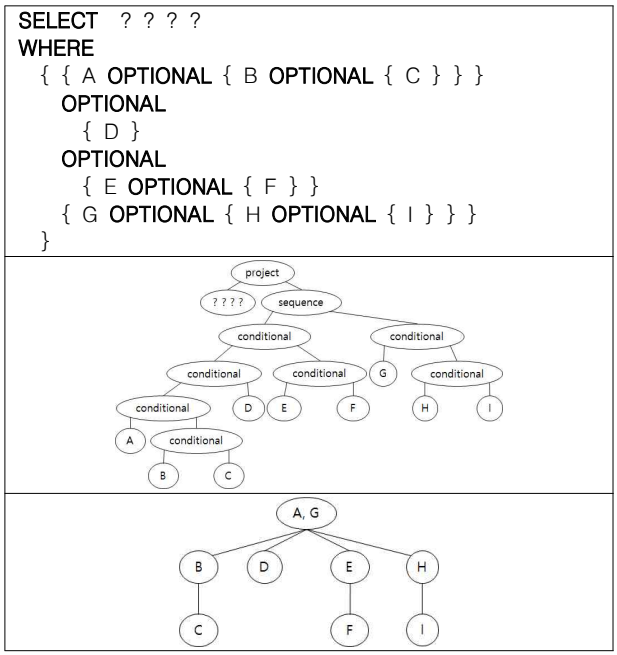

Fig. 13. A query with OPTIONAL, its SPARQL algebra expression tree and its BGP tree

그림 13은 그림 11과 12 알고리즘의 이해를 돕기 위한 예이다. 그림 13의 SPARQL 질의문은 OPTIONAL 연산자를 다수 포함하고 있다. 질의문의 A~I는 BGP들로서 트리플 목록을 단순화시킨 것이다. 그림 13의 두 번째 박스에 있는 트리는 SPARQL 대수식 트리이다. 대수식 트리의 conditional 노드는 SPARQL 질의문의 OPTIONAL 연산자에 대응한다. 그림 13의 세 번째 박스에 있는 트리는 BGP 트리로서 그림 11과 12의 알고리즘에 의해서 그림 13의 두 번째 박스에 있는 대수식 트리로부터 유도된 것이다.

BGP 트리의 루트 노드는 질의문의 모든 결과 튜플들이라면 반드시 만족시키고 있는 트리플 패턴들이 있는 곳이다. 루트 노드에 A와 G가 함께 있는 이유는 SPARQL의 기본적인 BGP 결합 논리가 conjunction이기 때문이다. BGP 트리의 루트 노드가 아닌 노드는 문자 그대로 옵셔널 BGP이다. 즉, 질의 결과 튜플들은 이 옵션 BGP들에 매칭될 수도 있고 그렇지 않을 수도 있다. 따라서 질의 결과 내의 임의의 튜플이 만족시킬 수 있는 BGP 집합은 BGP 트리의 임의의 노드로부터 루트 노드까지의 경로에 출현하는 BGP를 요소로 하는 집합들과 그 집합들의

가능한 모든 조합을 각각 합한 합집합들이다.

4. Explanation of Query Result with RDFa

질의 결과에 대한 문맥 정보는 질의 결과 내의 각각의 튜플에 대해서 생성된다. 질의 결과 튜플에 대한 문맥 정보를 생성하기 위해서는 다음의 절차를 따른다. 우선 BGP 트리 리스트에서 질의 결과를 설명할 수 있는 BGP 트리를 선정한다. BGP 트리는 대수 트리에서 유도되었고 대수 트리는 고유의 변수명에 대한 postfix를 가지고 있기 때문에 이를 근거로 적합한 BGP 트리를 선정한다. 다음으로 선정된 BGP 트리에서 질의 결과를 설명할 수 있는 트리플들을 수집하여 트리플 집합을 구성한다. 이 과정에서 BGP 트리의 BGP들은 변수를 포함하고 있으므로 변수를 질의 결과 튜플의 변수값으로 치환하는 작업이 동반된다. 마지막으로 수집된 트리플 각각을 RDFa로 표현하여 HTML 파일 내에 삽입시킨다.

4.1 Collecting Triples to Explain a Result Tuple

그림 14의 알고리즘은 BGP 트리의 루트 노드를 매개 변수로 하여 호출된다. 그림 14의 알고리즘은 깊이우선탐색 방식으로 BGP 트리를 탐색하며 질의 결과 튜플을 설명할 수 있는 트리플을 수집한다. 이 때, 트리의 탐색에 back tracking을 적용하기 위하여 그림 15의 promising 함수를 사용한다. promising 함수의 역할은 현재 방문한 노드 *v*에 있는 트리플을 추출할지 말지를 결정하기 위하여 사용된다. promising 함수가 유망하지 않음을 의미하는 false를 반환하면 방문한 노드 *v*와 그 자식 노드들은 트리플 추출에서 제외된다.

SPARQL 질의 결과에 있는 각각의 튜플은 모두 최소한 BGP 트리의 루트 노드에 있는 BGP들은 만족시키기 때문에 존재하는 것이다. 따라서 BGP 트리의 루트 노드에 있는 트리플들은 무조건 트리플 집합으로 추출된다. 이 때, 루트 노드의 트리플에 포함되어 있는 변수들이 모두 SPARQL 질의문의 SELECT 절에 나열되어 있지 않을 수 있다. 이러한 변수는 결과 튜플에 변수값이 없으므로 구체적인 값으로 치환할 수 없다. 그러나 그 구체적인 값을 질의 결과로부터 얻지 못할 뿐이지 그 값은 반드시 존재한다. 본 연구에서는 이러한 변수가 주어와 목적어일 경우는 그림 14의 12행에서와 같이 구체적인 값이 아닌 RDF blank 노드(bnode)로 치환하고 술어일 경우는 그림 14의 10행에서와 같이 'rdf:object' 프로퍼티로 치환한다.

BGP 트리의 루트 노드와는 달리 루트 노드가 아닌 노드의 BGP는 옵션 BGP이다. 옵션 BGP에 포함되어 있는 변수값의 존재여부는 질의 결과 튜플에 그 변수값이 존재해야만 알 수 있다. 또한 그 변수값의 존재여부를 판단할 수 있어야만 질의 결과 튜플이 그 변수를 포함한 트리플 패턴에 매치하는지 여부를 판단할 수 있다. 이러한 이유로 그림 15의 8~9행에서 옵션 BGP에 포함된 변수 *var*가 루트 노드에는 출현하지 않는 변수이며 그 값이 질의 결과 튜플에 존재하지 않을 경우 그 변수를 포함한 트리플 패턴을 질의 결과 튜플이 만족시키는지 여부를 판단할 수 없으므로 유망하지 않다고 판단하도록 하였다. 예를 들어, 그림 13의 BGP 트리에서 현재 방문한 노드가 B라 하고 B가 유망하지 않다면 노드 B에 있는 트리플들은 추출되지 않으며 다음 방문 노드는 D가 된다. 노드 C는 방문하지 않기 때문에 노드 C에 있는 트리플들은 추출되지 않는다. 만약 노드 B가 유망하지 않은 상태에서 질의 결과 튜플이 노드 C에 있는 BGP를 모두 만족시킨다 하더라도 그것은 질의 결과 튜플이 노드 B에 있는 BGP를 모두 만족시킨다는 전제에서만 의미가 있다. 왜냐하면 노드 C의 BGP는 노드 B의 BGP에 대한 옵션 BGP이기 때문이다. 이것이 노드 C를 방문하지 않는 이유이다.

```
01 void buildTripleSet(BGPNode v)
02 input: a node v in a BGP tree
03 if (promising(v)) {
04   for (each triple t in v ) {
05     t' ← t.clone() // t' is a copy of t.
06     for (each variable var in t' ) {
07       if (the value of var is in the result tuple)
08         Replace var with the value.
09       else if (var is used as a predicate)
10         Replace var with 'rdf:object'.
11       else
12         Replace var with a blank node.
13     } // end for
14     Add t' to the triple set.
15   } // end for
16   for (each child node u of v)
17     buildTripleSet(u)
18 } // end if
```

Fig. 14. Algorithm to collect triples from a BGP tree

```
01 boolean promising(BGPNode v)
02 input: a node v in a BGP tree
03 output: true or false
04 if (v is the root node) return true
05 else {
06   for (each triple t in v)
07     for (each variable var in t)
08       if (var dose not occur in the root node &&
09           the value of var is not in the result tuple)
10         return false
11 }
12 return true
```

Fig. 15. Promising function for a BGP tree

4.2 Generating RDFa Expression

트리플 집합에 있는 각각의 트리플에 대한 RDFa 표현을 생성해내면 질의 결과에 대한 문맥 정보를 생성하기 위한 모든 과정이 완료된다. RDF 트리플을 HTML 문서에 표현하기 위해서는 임의의 HTML 태그에 RDFa 속성을 추가해야 한다. RDFa는 하나의 트리플에 대한 다양한 RDFa 표현법을 제공한다. 본 연구에서는 트리플의 주어는 @about, 술어는 @property, 목적어는 @resource 혹은 @content 속성을 사용할 것이다. @resource 속성은 목적어가 IRI 혹은 bnode일 경우를 위한 것이며 @content 속성은 목적어가 리터럴일 경우를 위한 것이다.

그림 16은 그림 5의 질의 결과를 설명하는 RDFa 표현이다. 각각의 질의 결과 튜플은 태그로 구분된다. 첫 번째 영역에 있는 두 개의 <meta> 태그는 각각 그림 5에서의 트리플 ①과 ②를 RDFa로 표현한 것이며 두 번째 영역에 있는 두 개의 <meta> 태그는 각각 그림 5에서의 트리플 ③과 ④를 RDFa로 표현한 것이다. 그림 16의 bnode인 '_:x1'과 '_:x2'는 사용자의 원본 질의문에서 사용된 변수명 'x'와 질의 결과 튜플의 행번호 '1' 혹은 '2'를 결합한 것이다.

```
<div>
  <meta about="_:x1" property="rdf:type" resource="foaf:Person">
  <meta about="_:x1" property="foaf:name" content="Alice">
</div>
<div>
  <meta about="_:x2" property="rdf:type" resource="foaf:Person">
  <meta about="_:x2" property="foaf:name" xml:lang="en" content="Bob">
</div>
```

Fig. 16. RDFa expression for the query result of Fig. 5

그림 17은 그림 16의 RDFa 표현을 RDFa Play[22]에서 가시화한 결과이다. HTML에 포함되어 있는 RDFa 표현들은 그림 17과 같이 기계에 의해 RDF 그래프로 해석된다.

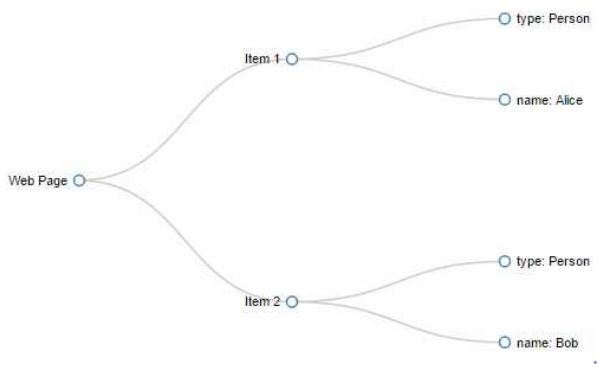

Fig. 17. Visualization of Fig. 16

IV. Evaluation

전술한 내용을 구현한 본 연구의 시스템은 JavaSE-1.7과 Apache Jena ARQ를 기반으로 한다. 본 장에서는 본 연구의 시스템을 BSBM(Berlin SPARQL Benchmark)[23]에 적용하여 평가한 내용을 기술한다. BSBM은 SPARQL 질의 엔진의 성능 평가를 위한 사양으로서 이를 위한 다양한 볼륨의 데이터 셋과 다양한 SPARQL 구문이 적용된 SPARQL 질의문으로 구성되어 있다. 평가를 위한 데이터 셋의 크기는 9.07 MB이며 BSBM에서 제공하는 데이터 생성기의 디폴트 설정으로 생성하였다. 생성된 데이터 셋은 Jena 기반으로 메모리에 적재된 상태로 서비스되도록 하였다. 평가를 위한 질의문은 BSBM에서 정의한 12개의 질의문 중에서 9번과 12번을 제외한 나머지 10개를 선정하였다. 본 연구는 SELECT 질의문을 처리 대상으로 하기 때문에 DESCRIBE 질의문인 9번 질의문과 CONSTRUCT 질의문인 12번 질의문을 제외하였다.

그림 18은 평가를 위한 각각의 질의문에 대한 결과의 특성을 요약한다. 그림 18의 1행은 질의문 번호이며 2행은 질의문 결과의 튜플 개수이며 3행은 질의 결과의 크기이다. BSBM의 모든 질의문은 %parameter% 와 같은 형식의 매개변수를 포함하고 있으며 이 매개변수들을 구체적인 값으로 치환한 후 사용해야 SPARQL 처리기가 이해할 수 있는 SPARQL 질의문이 된다. 본 연구에서는 매개변수 %parameter%을 SPARQL의 변수인 ?parameter와 같은 형식으로 수정하였다. 그 이유는 구체적인 값보다 변수로 치환하는 것이 더 많은 질의 결과 튜플을 유도할 수 있기 때문이다.

Query Number	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q10	Q11
Row Count	10	2375	0	10	5	74	2025	20	0	80351
Size (KB)	2	5,927	1	3	2	11	997	56	1	180,806

Fig. 18. Characteristics of query results for evaluation

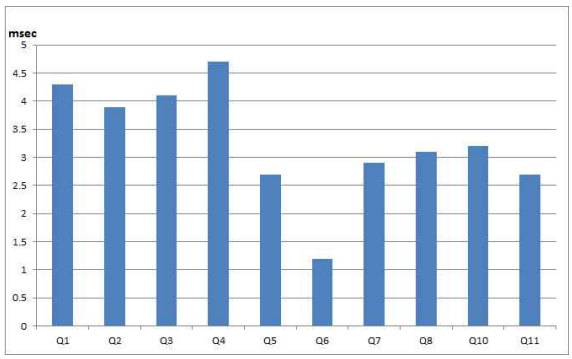

Fig. 19. Processing time before query execution

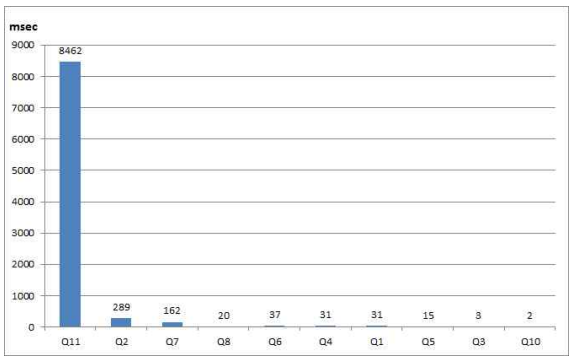

Fig. 20. HTML generating time

그림 19는 사용자의 원본 질의문을 입력받아 UNION 연산자에 의한 모호성 검사와 그에 따른 필요시 새로운 질의문의 생성 그리고 BGP 트리의 생성까지를 위한 처리 시간이다. 실험은 각 질의문에 대하여 20회씩 무작위 순서로 실행하였다. 그림 19는 이상치를 제외한 평균으로서 질의문의 종류와 무관하게 모두 수 밀리초 내에 처리함을 보여준다. 각 질의문 당 1회 정도 100~200 사이의 밀리초 결과가 있었으나 이상치로 판단하여 제외하였다. 각 질의문에 대하여 나타나는 시간 차이에 대한 분석은 유의하지 않다고 판단한다.

그림 20은 JSON 타입의 질의문 결과를 수신한 후 질의 결

과 내의 트리플을 RDFa 표현으로 번역하여 HTML 파일을 생성하기까지의 시간이다. 그림 20의 x축의 질의문 번호는 질의 결과 크기에 따라 내림차순으로 정렬하였다. 각각의 질의문에 대한 HTML 생성 시간은 질의 결과의 크기에 비례함을 보인다.

V. Conclusions

본 논문을 통해 SPARQL 질의 결과에 대한 RDFa 형식의 문맥 정보를 자동 생성하는 방법을 상세히 소개하였다. 질의 결과에 대한 문맥 정보의 근거는 SPARQL 질의문에 포함되어 있는 BGP이기 때문에 BGP 추출 전에 생성될 문맥 정보의 모호성을 제거하고 정확성을 제공하기 위하여 SPARQL 대수식 수준에서의 질의문 조작 작업을 수행하였다. 이 과정은 실세계에서 빈번하게 사용되는 질의문 패턴은 아니지만 복잡한 구조의 UNION 연산자와 OPTIONAL 연산자가 포함된 질의문도 처리할 수 있도록 설계하였다.

본 논문에서의 실험은 BSBM 사양을 이용하여 수행하였다. BSBM의 질의문들에는 ORDER BY, LIMIT, OPTIONAL, UNION, OFFSET, FILTER, REGEX, DISTINCT 등의 SPARQL 키워드들의 사용이 적절히 배분되어 있다. 실험 결과 이러한 구문이 포함된 질의문에서도 오류 없이 동작함을 확인하였다. 그러나 본 논문의 방법은 SPARQL 최신 사양인 SPARQL 1.1 [24]에서 새로 추가된 구문들에는 대응하지 못하는 한계가 있다. 따라서 향후 과제로서 SPARQL 1.1 질의문에도 대응할 수 있도록 그 기능을 확장해보고자 한다.

REFERENCES

[1] M. Schmachtenberg, C. Bizer, and H. Paulheim, "Adoption of the Linked Data Best Practices in Different Topical Domains," Proceedings of the 13th International Semantic Web Conference, pp. 245-260, 2014.

[2] D. W. Jo, and M. H. Kim, "Linked Legal Data Construction and Connection of LOD Cloud," Journal of the Korea Society of Computer and Information, Vol. 21, No. 5, pp. 11-18, May. 2016.

[3] D. W. Jo, and M. H. Kim, "A Framework for Legal Information Retrieval based on Ontology," Journal of the Korea Society of Computer and Information, Vol. 20, No. 9, pp. 87-96, Sep. 2015.

[4] E. Prud'hommeaux, and A. Seaborne, "SPARQL Query Language for RDF," W3C Recommendation, Mar. 2008.

[5] A. Seaborne, "SPARQL 1.1 Query Results JSON Format," W3C Recommendation, Mar. 2013.

[6] A. Seaborne, "SPARQL 1.1 Query Results CSV and TSV Formats," W3C Recommendation, Mar. 2013.

[7] S. Hawke, "SPARQL Query Results XML Format (Second Edition)," W3C Recommendation, Mar. 2013.

[8] Y. Theoharis, I. Fundulaki, G. Karvounarakis, and V. Christophides, "On Provenance of Queries on Semantic Web Data," IEEE Internet Computing, Vol. 15, No. 1, pp. 31-39, Jan. 2011.

[9] S. McCarron, B. Adida, M. Birbeck, G. Kellogg, I. Herman, and S. Pemberton, "HTML+RDFa 1.1 - Second Edition," W3C Recommendation, Mar. 2015.

[10] P. Buneman, S. Khanna, and W. C. Tan, "Data Provenance: Some Basic Issues," Proceedings of the Foundations of Software Technology and Theoretical Computer Science, pp. 87-93, 2000.

[11] S. Ram, and J. Liu, "A New Perspective on Semantics of Data Provenance," Proceedings of the First International Workshop on Role of Semantic Web in Provenance Managemant, 2009.

[12] H. Halpin, "Provenance: The Missing Component of the Semantic Web for Privacy and Trust," Trust and Privacy on the Social and Semantic Web Workshop at European Semantic Web Conference, 2009.

[13] P. Groth, and L. Moreau, "PROV-Overview: An Overview of the PROV Family of Documents," W3C Working Group Note, Apr. 2013.

[14] J. Zhao, and O. Hartig, "Towards Interoperable Provenance Publication on the Linked Data Web," Proceedings of the 5th Linked Data on the Web Workshop at the 21th International World Wide Web Conference, 2012.

[15] DCMI Metadata Terms, *http://dublincore.org/documents/dcmi-terms/*

[16] C. V. Damásio, A. Analyti, and G. Antoniou, "Provenance for SPARQL queries," Proceedings of the 11th International Conference on The Semantic Web, pp. 625-640, 2012.

[17] F. Geerts, T. Unger, G. Karvounarakis, and V. Christophides, "Algebraic Structures for Capturing the Provenance of SPARQL Queries," Journal of the ACM, Vol. 63, No. 7, Mar. 2016.

[18] R. Hasan, “Predicting query performance and explaining results to assist Linked Data consumption,” PhD Thesis, Université Nice Sophia Antipolis, Nov. 2014.

[19] B. Adrian, J. Hees, I. Herman, M. Sintek, and A. Dengel, “Epiphany: Adaptable RDFa Generation Linking the Web of Documents to the Web of Data,” Procedings of the 17th International Conference on Knowledge Engineering and Knowledge Management, pp. 178-192, 2010.

[20] A. Khalili, and S. Auer. "WYSIWYM-Integrated Visualization, Exploration and Authoring of Semantically Enriched Un-structured Content," Semantic Web Journal, Vol. 6, No. 3, pp. 259-275, 2015.

[21] Apache Jena, *http://jena.apache.org/index.html*

[22] RDFa Play, *https://rdfa.info/play/*

[23] Berlin SPARQL Benchmark Specification - V2.0, http://wifo5-03.informatik.uni-mannheim.de/bizer/berlinsparqlbenchmark/spec/20080912/index.html#completeQM

[24] S. Harris, and A. Seaborne, "SPARQL 1.1 Query Language," W3C Recommendation, Mar. 2013.

Authors

Ji Woong Choi received the B.S., M.S. and Ph.D. degrees in Computer Science and Engineering from Soongsil University, Korea, in 2001, 2003 and 2011, respectively.

Dr. Choi joined the faculty of the School of Computer Science and Engineering at Soongsil University, Seoul, Korea, in 2013. He is currently an Assistant Professor in the School of Computer Science and Engineering, Soongsil University. He is interested in information system.