工學碩士 學位論文

온톨로지 저장소에서의 그래프 레이블링을 이용한 효율적인 트랜지티브 클로저 질의 처리

Efficient Processing of Transitive Closure Queries in Ontology Store using Graph Labeling

2005年 2月

서울大學校 大學院

전기·컴퓨터 工學部

金 鍾 男

온톨로지 저장소에서의 그래프 레이블링을 이용한 효율적인 트랜지티브 클로저 질의 처리

Efficient Processing of Transitive Closure Queries in Ontology Store using Graph Labeling

指導敎授 金 炯 周

이 論文을 工學碩士 學位論文으로 提出함

2004年 10月

서울大學校 大學院

전기·컴퓨터 工學部

金 鍾 男

金鍾男의 工學碩士 學位論文을 認准함

2004년 12월

委 員 長 이 석 호 印

副委員長 김 형 주 印

委 員 이 상 구 印

요 약

온톨로지는 특정 개념에 대한 부가정보 및 개념간의 관계를 기술하는 방법으로서 고차원의 웹과 서비스를 실현하기 위한 시멘틱 웹, 그리고 지식관리 시스템을 비롯한 다양한 응용분야의 요구와 관심이 증가하면서 그 중요성이 대두되고 있다. 온톨로지에서 정보에 대한 접근은 특정 개념과 특정 관계를 가지는 데이터를 찾는 것이 주를 이루는데, 이러한 관계가 주로 트랜지티브 관계이기 때문에 트랜지티브 정보를 처리하는 것이 많은 비중을 차지한다. 또한 이와 같은 트랜지티브 클로저 질의 처리는 재귀 호출의 형태로서 그 처리 비용 또한 매우 크다.

본 논문에서는 이와 같은 트랜지티브 클로저 질의의 효율적 처리를 지원하기 위한 방법으로써 그래프 레이블링을 이용하여 전처리 하는 기법을 제안한다. 이 기법은 저장 공간을 효율적으로 사용하고 알고리즘도 단순한 특징을 가지기 때문에 이행적 질의에 대한 응답 시간을 줄이는 장점을 가지게 된다. 그리고 이와 같이 제안한 기법에 대해 기존 시스템들과 비교해 봄으로써 그래프 레이블링을 이용한 기법이 대용량 온톨로지에서의 트랜지티브 클로저 질의 처리에 효과적임을 보이고자 한다.

키워드: 온톨로지, 트랜지티브 클로저, 레이블링

Ontology, Transitive Relationship, Transitive Closure

학번: 2003-21557

목 차

1. 서 론

1.1 연구 동기

시맨틱 웹은 의미의 연결(Relationship between Web Resources)을 통해 정보를 제공한다. 즉, 정보의 의미를 개념으로 정의하고 개념 간의 관계성을 표현함으로써 정보가 공유된다. 이러한 시맨틱웹의 비전을 현실화하기 위해서는 온톨로지를 기반으로 의미 정보로 구성된 시맨틱 문서(Semantic Documents)들에 대한 추론을 통해 웹상에 존재하는 수많은 정보 간의 관계성을 밝혀내고 사용자가 필요로 하는 지식을 검색하기 위한 기술이 필수적으로 요구된다. 다시 말해서 온톨로지의 중요한 역할 중의 하나는 개념과 개념 간의 관계를 정의 하는 것이다. 대표적으로는 상속과 부분-전체 관계를 들 수 있는데, 이러한 관계와 Description Logic 을 이용하면 온톨로지에 직접적으로 표현되지 않은 개념 간의 관계를 추론을 통하여 제시하거나 이에 대한 질의에 답할 수 있게 된다. 시맨틱 웹에서는 자연어 위주의 기존 웹 문서와 달리 컴퓨터가 해석하기 쉽도록 의미를 부여한 계층을 가지고 있기 때문에 자동화된 에이전트나 정교한 검색 엔진들이 부여된 의미를 이용하여 고 수준의 자동화와 지능화를 이룰 수 있게 된다.

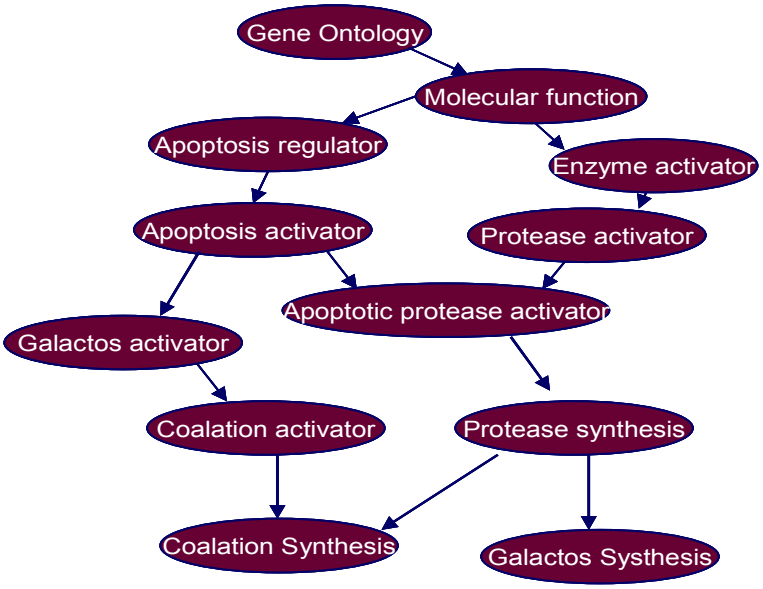


그림 1 Gene Ontology 의 예

웹의 발전과 더불어 온톨로지라는 용어를 자주 대하게 되는데 시맨틱 웹에서의 온톨로지는 '개념화의 규정'(specification of conceptualization)을 말한다. 온톨로지는 지식 표현의 한 형태로서, 한 도메인을 모델링 하는데 있어서 각 개념들을 추출하고 그 개념들의 연관 관계를 규정함으로써 해당 개념을 구체화 시키는 방법이다. 이러한 의미에서 어휘사전(thesaurus)이나 데이터베이스 스키마, 또는 야후(Yahoo)의 주제 분류와 같은 계층적 카테고리도 온톨로지의 일종으로 볼 수 있다. 온톨로지는 이러한 단순한 개념의 분류와 단순한 클래스-하위 클래스 관계 외에도 개념과 개념간의 다양한 관계와 개념의 속성과 제약조건, 그리고 추론 규칙을 가지게 되고, 온톨로지를 구성하는 방법에는 객체 지향 방법론이 응용되어

사용된다. 그 중에서 본 논문의 주제와 밀접한 관계인 subClassOf 나 subPropertyOf 는 트랜지티브 관계(transitive relationship)이고, 질의 처리시 해당 개념(concept)에 대한 인스턴스(instance)뿐 아니라 그 하위 클래스들도 모두 대상에 포함되게 된다. 즉 원래의 아니라 그 하위 클래스들도 모두 대상에 포함되게 된다. 즉 원래의 그래프 G= (N, E)에 대한 트랜지티브 클로저(transitive closure) G^+=(N, E^+)에 대해 질의를 하게 된다. 이론적으로, 어떤 그래프의 트랜지티브 클로저를 구하는 것은 해당 그래프에 대한 인접 행렬 표현을 생각했을 때 행렬의 곱의 복잡도에 해당하는 것으로, 이는 dynamic programming 으로 $O(n^3)$의 시간 복잡도(time complexity)를 갖고 개선된 알고리즘으로도 $O(n^2)$ 의 비용은 피할 수 없다. 그리고 대용량의 데이터를 가정했을 때 이를 동적으로 계산하기 보다는 미리 계산(pre-computing) 해 놓은 후 참조하는 방식이 일반적이다.

온톨로지 모델에 대한 트랜지티브 클로저 질의(transitive closure queries)는 크게 세가지 종류이다.

1. 두 개념이 상속 관계인지 체크
2. 어떤 개념의 모든 조상/후손 개념을 구함
3. 두 개념의 NCA (Nearest Common Ancestor)를 구함

표 1 온톨로지 질의 벤치마크(ODP 카탈로그를 RDFSuite 에 저장했을 때)

Query	Description	Time
Q1	Find the range(or domain) of a property	0.0012
Q2	Find the direct subclasses of a class	0.0124
Q3	Find the transitive subclasses of a class	341.98
Q4	Check if a class is a subclass of another class	0.0662
Q5	Find the direct extent of a class(or property)	0.027
Q6	Find the transitive extent of a class(or property)	482.45
Q7	Find if a resource is an instance of a class	0.00174
Q8	Find the resource having a property with a specific values	0.0466
Q9	Find the instances of a class that have a given property	0.1059
Q10	Find the properties of a resource and their values	0.0076
Q11	Find the classes under which a resource is classified	0.0015

이러한 트랜지티브 클로저 질의는 재귀순환(recursion)의 형태로서 대부분의 SQL 구현들이 이러한 재귀적 표현을 지원하지 못하므로, 이를 계속적인 SQL 문장으로 반복적으로 구해야 하고 이것은 시간 측면에서 성능의 큰 저하를 가져온다. 또한 현재 Jena[5]와 RDFSuite[4]을 포함한 일반적인 온톨로지 저장소들은 트랜지티브 클로저 질의를 보편적인 그래프 탐색과 메모리 캐쉬에 의존하고 있어서, 대용량의 온톨로지를 가정했을 때 트랜지티브 클로저 질의에 대한 성능이 좋지 못하다. 또한 Gene Ontology 시스템에서는 graph_path 테이블에 스키마 그래프의

트랜지티브 클로저를 미리 계산하여 저장하고 있으나 이는 공간 측면에서 비효율적 일 뿐 아니라 원래의 트리플(triple)과 그 데이터에서 유추된 트리플을 혼합하여 저장하므로 바람직하지 못하다. 또한 gene ontology 처럼 온톨로지 내의 상속관계가 is_a, part_of 등 여러 개 일 수 있고, 관계(relationship)가 점차 세분화(specialization) 될 수 있는데 온톨로지 저장소들은 기본적으로 데이터를 트리플 형태로 저장하므로 결국 세분화된 관계에 대해서는 더 많은 트리플 간의 조인 연산이 필요하게 된다.

일반적으로, 온톨로지에서의 효율적인 Transitive Closure 질의 기법은 다음 세 가지의 서로 배반적인 요건들을 만족시켜야 한다.

1. 저장공간 효율성: 대용량 온톨로지를 가정
2. 검색 효율성: 최소한 평균 케이스(average case)에 대해서 효율적인 검색이 보장 되어야 함.
3. 갱신 효율성: 점진적(incremental)으로 갱신하는 비용이 다시 재계산하는 비용보다는 적어야 함.

본 논문은 다음과 같은 의미(contribution)를 가진다.

1. 구간 기반 레이블링을 온톨로지에 적용하여 클래스(class)나 속성(property)들간의 상속관계를 레이블만 보고도 결정할 수 있게 한다. 일반적으로 그래프의 트랜지티브 클로저는 O(n2)의 공간을 차지하나 구간 기반 레이블링 기법은 O(n)으로 필요한 정보를 유지 할 수 있다.

2. 생물 정보학에서의 사실상의 표준 온톨로지를 구현한 Gene Ontology 시스템에서의 트랜지티브 클로저 질의에 대한 처리 방법과 널리 쓰이는 온톨로지 저장소인 Jena 시스템의 처리 방법을 설명하고 대용량의 온톨로지를 가정했을 때 이의 문제점을 분석한다.
3. 관계형 모델에 비해 비교적 변화가 자주 일어나는 온톨로지 모델에서 구간 기반 레이블링의 점진적(incremental)인 갱신이 어떻게 가능한지 설명한다.
4. 대용량 온톨로지의 예로써 Gene Ontology 에 대한 실험을 통해 기존의 기법과 본 논문의 기법을 비교하고, 제안하는 기법의 우수성을 입증한다.

1.2 논문의 구성

본 논문의 구성은 다음과 같다.
2 장에서는 관련연구에 대해 살펴보고, 3 장에서 본 논문에서 사용되는 데이터 모델에 대해 설명하고 기존의 기법의 문제점을 설명한다. 4 장에서는 본 논문이 제안하는 기법에 대해 알아본다. 5 장에서는 실험을 통해 효율성을 설명하고, 6 장에서는 결론 및 향후 연구를 기술한다.

2. 배경 지식

2.1 에서는 Gene Ontology 의 데이터 모델에 대해 설명하고, 2.2 에서는 표준 웹 온톨로지로서의 OWL 에 대해 알아본 후 2.3 에서 시멘틱 웹 에서의 일반적인 추론에 대해 설명한다.

2.1 Gene Ontology 데이터 모델

생물학 연구로 인해 쏟아내는 정보의 양이 많아지면서 적은 수의 DB 로는 그것들을 모두 포용할 수 없게 되었고 데이터의 특성과 DB 의 목적에 따라 수많은 생물학 관련 DB 들이 만들어져 왔다. 이러한 DB 들은 서로 독립적인 것이 아니라 정보의 의미상 연관관계에 있기 때문에, 이들간 정보의 공유와 교환은 필연적이라고 할 수 있다. 그러나 서로 다른 용어와 문법의 사용으로 인해서 정보의 흐름이 원활하지 못하다. 그렇기 때문에 생물정보학에서 DB 정보의 공유와 교환문제는 가장 우선적으로 정립되어야 하는 문제이다. 또한 다른 이유에서도 용어의 정리는 필요하다. 생명유지에 기본적인 단백질이나 유전자, 그리고 그것에 관련된 기능들은 하나의 종에만 국한된 것이 아니라 여러 종에 공통적으로 있을 수 있다. 그렇기 때문에 관련된 모든 종에서 공통적으로 쓰일 수 있는 용어의 확립이 중요하고, 예외사항이 있다면 그에 대한 규정들을 세워야 한다. 온톨로지란 철학의 '존재론'을 뜻한다. 즉 생물정보학에서 말하는 온톨로지란 것은 크게

볼 때, 유전자들의 복잡한 네트워크 중에서 하나의 유전자의 위치를 확립시키는 것이라 볼 수 있는 것이며, 이렇게 여러 종 사이에서 쓰이는 용어의 확립 문제도 온톨로지라 할 수 있는 것이다. Gene Ontology Consortium 의 목적은 다양하게 쓰이고 있는 생물학 용어들을 정리하고, 복잡한 유전자 네트워크를 DB 나 응용 프로그램들 사이에서 통용될 수 있는 문서 형태로 전환하는 데에 있다. Consortium 에 참여하는 그룹들은 그림 1 과 같다.

그림 2 Gene Ontology Consortium 참여 그룹

초반에는 FlyBase 를 주축으로 하여 몇 그룹 되지 않았지만, 현재 그 수가 많이 늘어난 상태이고, 참여 그룹은 더욱 늘어날 것으로 보인다. 이러한 DB 들은 다음과 같은 세가지 방법으로 Gene Ontology(GO)와 합작을 하게 된다.

1) GO terms 과 자신의 DB 사이의 database cross-link 를 만든 뒤에, GO link table 을 제공한다.
2) GO terms 을 이용하여 질의를 할 수 있도록 한다.
3) GO databases 에 있는 terms 의 확장이나 재정의 등으로 직접 개발에 기여한다.

Gene Ontology 는 용어(term)를 크게 Molecular Function, Biological Process, Cellular Component 세 가지로 분류하고 있다. 이 세가지를 설명하기에 앞서서 우선 Gene Product 에 대한 정의를 명확하게 하자면, Gene Product 는 물리적인 것에 국한한다. 다시 말해서 단백질(Protein) 또는 RNA 등일 것이다. 예를 들면 alpha-globin, small ribosomal RNA 등이다. 복합체(complex)를 이루어 작용하는 gene products 를 gene product groups 이라 한다. gene product group 은 gene products 뿐만 아니라, heme 와 같은 작은 molecules 도 포함한다. Molecular Function 은 Gene Product 가 잠재적으로 갖는 능력을 뜻한다. 즉 실제 어디서 언제 쓰이는지에 대한 구체화 없이, 무엇을 할 수 있는지에 대한 내용을 갖게 된다. Broad functional term 을 예로 들면 'engyme', 'transporter', 'ligand' 등이 있을 수 있고, narrower functional terms 을 예로 들면, 'adenylate cyclase' 또는 'toll receptor ligand' 등이 있을 수 있다. 그림 2 는 molecular function 에 대한 것이다.

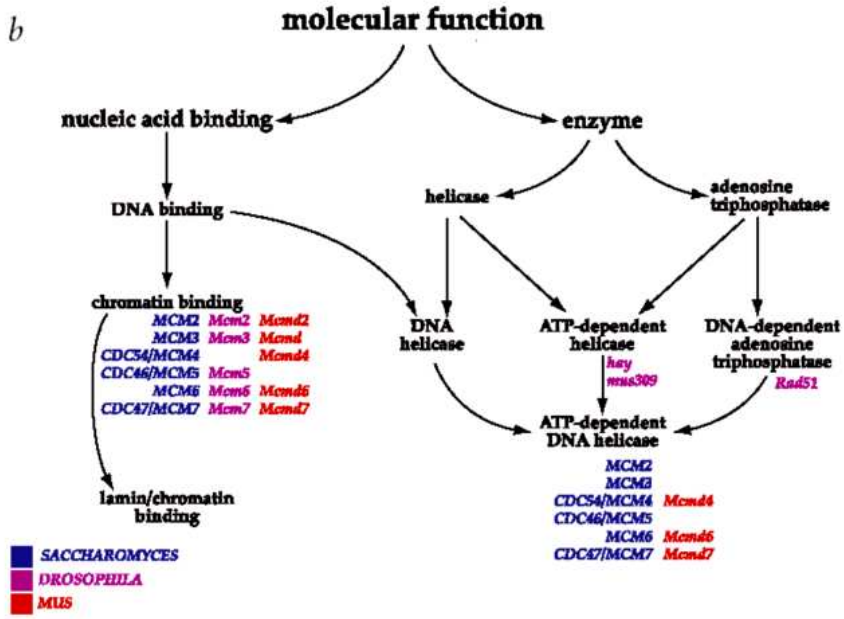


그림 3 Molecular Function 의 예

Biological Process 는 생물학 적인 목적을 뜻한다. 즉 하나 또는 그 이상의 Molecular function 이 순서대로 조합되어 이루어 내는 것이다. Broad biological process terms 으로 예를 들면 'cell growth and maintenance' 또는 'signal transduction'이 있고 specific terms 을 예로 들면 pyrimidine metabolism', 'camp biosythesis' 등이 있다. Biological Process 는 Molecular Function 과 의미상으로 구분하는 것이 어려울 수도 있는데, GO 에서는 1 개 이상의 구별되는 스텝이 있어야만 process 로 인정하도록 일반적인 규칙을 정하고 있다. 그림 3 은 DNA metabolism 을 표현하는 biological process 의 일부를 표시하는 예이다.

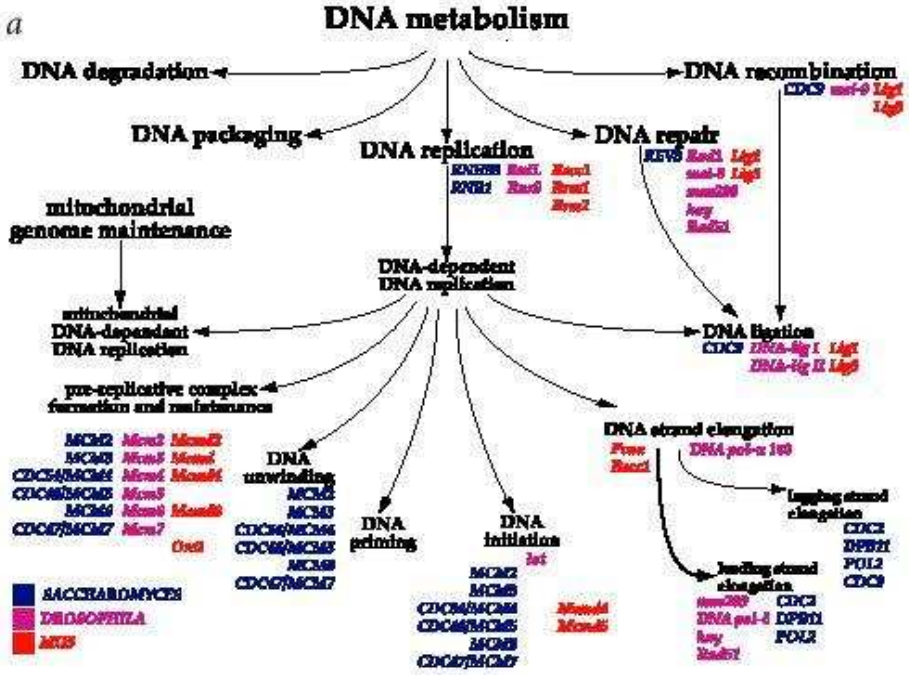


그림 4 Biological Process 의 예

Cellular Component 는 세포의 구성원을 말하는 것으로써 해부학적인 구조(anatomical structure)에 대한 명명이 주가 된다. ‘rough endoplasmic reticulum’ 또는 ‘nucleus’ 등이 그것이고 gene product group 의 예를 들면 ‘ribosome’, ‘proteasome’ 등이 될 것이다.

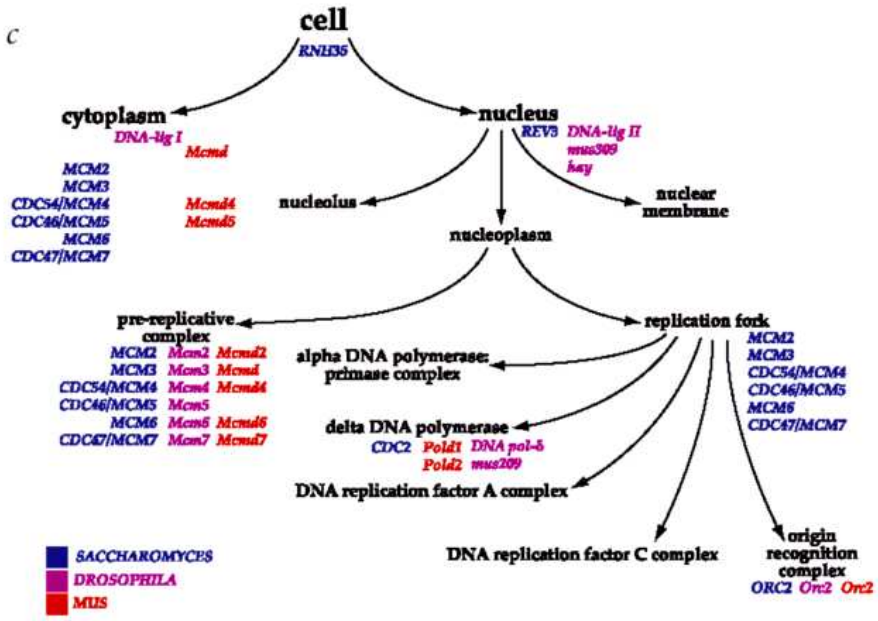


그림 5 Cellular Component 의 예

Function, Process, Component 등은 전체적으로 비순환 방향 그래프(DAG)로 표현된다. DAG 는 자식 노드가 여러 개의 부모 노드를 가질 수 있다는 것이 특징이고 (multiple inheritance), 하나의 자식 개념(child term)은 자신의 부모 개념(parent term)에 대한 인스턴스(instance)이거나 (is_a relationship) 또는 컴포넌트(component)이다 (part_of relationship). 자식 노드는 하나 이상의 부모를 갖고, 서로 다른 부모에 대해 다른 클래스의 관계를 갖게 될 것이다. 표준적인 데이터 모델로 각광을 받고 있는 XML 만으로는 DAG 구조를 효율적으로 표현하기 힘들기 때문에, Gene Ontology 는 rdf link 등의 RDF model 을 사용하고 있다.

2.2 표준 웹 온톨로지로서의 OWL

현재 우리가 사용하고 있는 웹은 인터넷 상에서 사용자에게 컨텐츠를 제공하는 정보 시스템인 반면에, 시맨틱 웹은 사람뿐만 아니라 컴퓨터(기계)가 직/간접적으로 처리할 수 있는 데이터를 위한 웹을 의미한다. 따라서 시맨틱 웹을 위해서는 사람이 직관적 또는 의미적으로 판단하여 처리하는 부분을 컴퓨터가 처리 할 수 있도록 사용되는 용어(vocabulary)에 대한 공통적인 합의가 필요하게 되며, 이를 웹 온톨로지라 부른다.

온톨로지는 철학의 '존재론'으로부터 그 기원을 찾아볼 수 있다. 고대 그리스의 존재론은 나를 포함한 이 세계가 어떻게 구성되어 있는가에 대한 문제 해결을 추구하였으며, 자연 법칙이나 실제 형태를 포함한 존재(being)에 관한 문제를 다룬다. 즉, 구성에 대한 정의를 하기 위해서는 용어에 대한 공통적인 표현이 필요하게 되며, 구성 개체간의 관계들에 대한 표현 및 의미를 부여할 수 있어야 했던 것이다. 존재론을 웹과 연관지어 살펴보면, URI(Uniform Resource Identifier)로 표현되는 자원은 모두 웹의 구성 개체가 되며, 구성 개체에 대한 정의 및 개체간의 관계들이 필요하게 된다. 그러나 현재의 HTML(Hypertext Markup Language) 기반의 웹은 시각적 효과를 위한 정보만을 컴퓨터에게 제공할 뿐 그 정보의 의미에 대한 정보는 제공하지 않는다. 따라서 W3C 에서는 XML 및 RDF 를 기반으로 온톨로지 표현 언어를 설계하고 있다. 그러나 W3C 의 활동에서는 존재론에서 거론하는 모든 형상에 대한 표현은 실로 매우 어렵기 때문에 웹이라는 특정 분야 안에서(구체적으로는

각 응용 도메인 별의 국한된 범위 내에서)의 표현만 가능하도록 웹 온톨로지 언어를 설계하고 있다. 초기의 온톨로지는 단순히 지식 공유와 재사용을 위해서 개발되었다. 최근에는 온톨로지의 개념이 지능적인 정보 통합, 협동 정보 시스템, 정보 검색, 전자 상거래, 지식 관리 같은 분야로 확대되고 있다. 또한 사람과 컴퓨터 사이에서 공유되고, 공통된 이해를 필요로 하기 때문에 온톨로지의 개념이 더욱 대중화되고 있다.

또한 온톨로지는 WWW 에서 적용되는 시맨틱 웹을 가능하게 한다. 그리고 응용 프로그램 사이에서 웹 기반 지식 처리, 공유, 재사용 하는 것을 가능하게 하는 중요한 역할과 함께 사람과 응용 프로그램 시스템 사이에서 공통된 주제의 의사소통을 제공하여, 전자 상거래에서 구매자와 구입자간의 기계기반의 의사소통을 가능하도록 만들어 준다. 예로, 검색 엔진에서 찾으려고 하는 키워드와 의미적으로 유사하지만 구조적으로 다른 단어들을 가진 웹 페이지들을 찾아주는 역할을 한다.

시맨틱 웹의 온톨로지 표준화는 W3C 를 중심으로 진행되고 있다. W3C 는 2001 년 웹 온톨로지 워킹그룹(Web-Ontology Working Group)을 구성하여 표준화 및 기술 개발을 진행하고 있다.

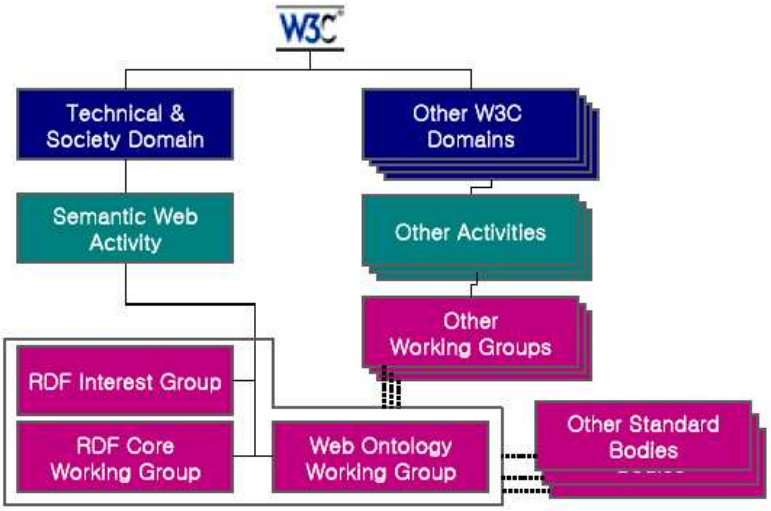


그림 6 W3C 의 웹 온톨로지 관련 구조

웹 온톨로지 워킹 그룹은 2002 년에 웹 온톨로지 언어(OWL: Ontology Web Language)의 필요사항에 대한 초안 버전 1.0 을 발표 한 것을 시작으로 웹 온톨로지 언어에 대한 구체적인 표준화를 진행 하고 있다. 더불어, 온톨로지를 위해서 메타데이터(Metadata)는 필수 요소로서 생각해야 하며, 이는 온톨로지가 의미론 적으로 상호 운용성(inter-operability)을 가져야 하기 때문이다. DCMI (Dublin Core Metadata Initiative) 에서는 메타 데이터의 표준화를 진행하여, 더블린 코어(Dublin Core)와 W3C 의 RDF(Resource Description Framework)를 표준으로 채택하였다.

마크업 언어 표준으로는 초기에 CycL(CYCR Language), KIF(Knowledge Interchange Format), Ontolingua 라는 마크업

언어가 연구되었으나 온톨로지를 표현하는 것이 용이하지 않고, 기존의 HTML, XML 과의 연계성이 떨어져 실패하였다. 시맨틱 웹을 위한 마크업 언어는 온톨로지를 표현하고 의미에 대한 정확한 정의와 데이터간의 논리적인 관계 설정 등의 기능을 가지고 있으며, 마크업 언어 기반의 에이전트를 이용하여 데이터간의 관계를 통한 정보추출과 추론이 가능해야 한다. 대표적인 인공지능 기반의 마크업 언어는 OIL(Ontology Inference Layer), DAML(DARPA Agent Markup Language), SHOE(Simple HTML Ontology Extensions)와 최근에 W3C 에서 개발중인 Notation 3(N3)가 있다. OIL 과 DAML 은 RDF/RDF 스키마를 기반으로 연구가 진행 되고 있는 마크업 언어이며, 현재 유럽 연합의 IST(Information Society Technologies)를 중심으로 연구 진행 중에 있으며 OIL 에 대한 백서를 발표한 상태이다.

DAML 은 미국 국방 과학 연구소(DARPA)의 지원을 받아 연구되었던 언어로 소프트웨어 에이전트간의 통신을 가능하게 한 언어이지만, 초기에는 온톨로지를 표현을 하지 못하는 문제점이 있었다. DAML 은 온톨로지를 표현하기 위해 OIL 을 포함하게 되었으며, DAML+OIL 로써 연구가 진행되어 왔으며 현재의 OWL 의 모체가 되고 있다. SHOE 는 메릴랜드 대학을 중심으로 연구가 되고 있는 언어로써, HTML 에서 온톨로지를 표현이 가능한 언어이이며, 현재 버전 1.01 의 발표하였다. N3 는 RDF 와 로직, 그리고 데이타를 하나의 언어로 표현하기 위한 최적화된 방법을 제공하이 위한 언어이다. 또한 N3 는 로직 언어를 이해하고 있다면, 쉽게 접할 수 있도록 설계되어 있고, 사용자 중심으로 설계된

언어이다. N3 는 RDF 를 대치하기 위한 것은 아니며, RDF 와 매핑을 통하여 상호보완적 역할을 수행하는 것을 그 목적으로 하고 있다.

OWL 은 문서에 포함된 정보를 어플리케이션을 이용하여 자동 처리하고자 할 때 활용하는 언어이다. OWL 을 이용하면 임의의 어휘를 구성하는 용어(term)의 의미와 용어들 간의 관계를 명시적으로 표현할 수 있다. 이와 같이 용어와 용어들 간의 관계를 표현한 것을 온톨로지(Ontology)라 한다. OWL 은 XML, RDF, RDF-S 보다 더 많은 의미 표현 수단을 제공하므로, 웹 상에서 기계가 해석할 수 있는 컨텐트를 작성하는데 있어 이들 언어보다 뛰어나다. OWL 은 DAML+OIL 웹 온톨로지 언어로부터 파생된 언어이다. OWL 에는 DAML+OIL 의 설계 및 활용 경험으로부터 습득된 지식이 반영되어 있다.

아래의 OWL 어휘들은 많이 사용되는 RDF 스키마와 관련된 어휘들이다.

1) Class

클래스는 동일한 속성을 지니고 있어 하나의 부류로 모아지는 개체들의 군(群, group)을 정의한다. 예를 들어, Deborah 와 Frank 는 둘 다 사람이므로 Person 클래스에 속한다. 클래스들은 subClassOf 를 통해 특성화 계층(specialization-hierarchy)으로 구조화될 수 있다. OWL 에는 사전 정의된 클래스로서 Thing 과 Nothing 이 있다. Thing 은 계층 구조 상 가장 일반화된 클래스로서

모든 OWL 클래스의 상위 클래스(superclass)이고, 모든 개체(individual)를 멤버로 가진다. 반대로 Nothing 은 모든 OWL 클래스의 하위 클래스(subclass)이고, 멤버를 가지지 않는다.

2) rdfs:subClassOf

임의의 클래스가 다른 클래스의 하위클래스임을 선언하는 문장을 기술함으로써 클래스 계층 구조를 구축할 수 있다. 예를 들어, Person 클래스를 Mammal 클래스의 하위클래스로 선언할 수 있다. 이를 바탕으로 추론기는 어떤 개체가 Person 클래스에 속하면, 그 개체가 Mammal 클래스에도 속한다는 사실을 유추해 낼 수 있다.

3) rdf:Property

속성은 개체 간의 관계 및 개체와 데이터 값 사이의 관계를 표현한다. hasChild, hasRelative, hasSibling, hasAge 등은 속성의 몇 가지 예이다. 처음 세 속성은 Person 클래스 인스턴스 간의 관계를 표현하기 위해 사용할 수 있으며(이와 같은 속성을 ObjectProperty 라 한다.), 마지막 속성(hasAge)은 Person 클래스의 인스턴스와 정수 데이터 값 사이의 관계를 표현하기 위해 사용할 수 있다(이와 같은 속성을 DatatypeProperty 라 한다.). owl:ObjectProperty 와 owl:DatatypeProperty 는 모두 RDF 클래스인 rdf:Property 의 하위 클래스이다.

4) rdfs:subPropertyOf

임의의 속성이 한 개 또는 여러 속성의 하위속성임을 선언하는 문장을 기술함으로써 속성 계층 구조를 구축할 수 있다. 예를 들어, hasSibling 속성을 hasRelative 속성의 하위속성으로 선언할 수 있다. 이로 부터, 추론 시스템은 임의의 두 개체가 hasSibling 으로 관계되어 있으면 두 개체 사이에 hasRelative 관계도 존재함을 유추해 낼 수 있다.

5) rdfs:domain

속성의 정의역(Domain)은 속성을 적용할 수 있는 주의할 점은 rdfs:domain 이 전역적인 제약 사항을 기술한다는 점 개체의 집합을 제한한다. 한 개체가 어떤 속성을 통해 또 하나의 개체와 관련되어 있다면, 그 개체는 속성의 정의역으로 선언된 클래스에 속해야 한다. 예를 들어, Mammal 클래스가 hasChild 속성의 정의역으로 선언되어 있고 "Frank hasChild Anna"라는 관계 선언이 주어지면, 추론기는 Frank 가 Mammal 클래스에 속한다는 사실을 유추할 수 있다. 여기서 이다. 왜냐 하면, 정의역 선언은 특정 클래스에 연계하여 속성에 대해 기술하는 것이 아니라 속성 자체의 특성으로 선언하기 때문이다. 제약 사항의 범위에 대하여는 아래의 속성 제약 관련 어휘들을 참조한다.

6) rdfs:range

속성이 값으로 가질 수 있는 개체의 집합을 제한한다. 한 개체가 어떤 속성을 통해 또 하나의 개체와 관계되어 있다면, 후자 개체는

속성의 공역(range)으로 선언된 클래스에 속해야 한다. 예를 들어, Mammal 클래스가 hasChild 속성의 공역으로 선언되어 있고 "Louise hasChild Deborah"라는 관계 선언이 주어지면, 추론기는 Deborah 가 Mammal 클래스에 속한다는 사실을 유추할 수 있다. rdfs:domain 과 마찬가지로 rdfs:range 는 전역 제약 사항이다. 제약 사항의 범위에 대하여는 아래 제시된 AllValuesFrom 등의 지역 제약 사항(local restriction)들을 참조한다.

7) Individual

개체는 클래스의 인스턴스이며, 개체들은 속성을 통해 상호 관계를 맺는다. 예를 들면, Person 클래스의 인스턴스로서 Deborah 라고 명명된 개체를 생성하고 hasEmployer 속성을 이용하여 StanfordUniversity 개체와 관계를 선언할 수 있다.

2.3 Semantic Web 추론 기법

현재 시맨틱 웹에 적용하려고 시도하고 있는 추론 기술은 논리기반(Logic based)의 추론과 룰 기반(Rule based)의 추론이 있다. 사실들과 일반적 규칙에 근거하여 추론하는 논리기반의 추론 엔진을 시맨틱 웹에 적용하려면, 예들 들면 FOL 기반 추론은 단조 추론(monotonic reasoning)만 가능하다거나, true/false 값만 대상으로 다루어야 한다거나, 또는 학습(learning) 기능이 없다는 것과 같은 몇 가지 단점들이 있다. 또한 룰 기반의 추론은 똑같은

문제가 다시 주어져도 똑같은 양의 작업(추론)을 하여 같은 결론을 얻거나, 룰 시스템에서 제공하는 지식의 영역을 벗어나는 문제에 대해서는 전혀 도움을 줄 수 없다든가, 또한 룰 시스템을 구축하고 유지하는데 많은 시간과 노력이 든다는 단점이 있다.

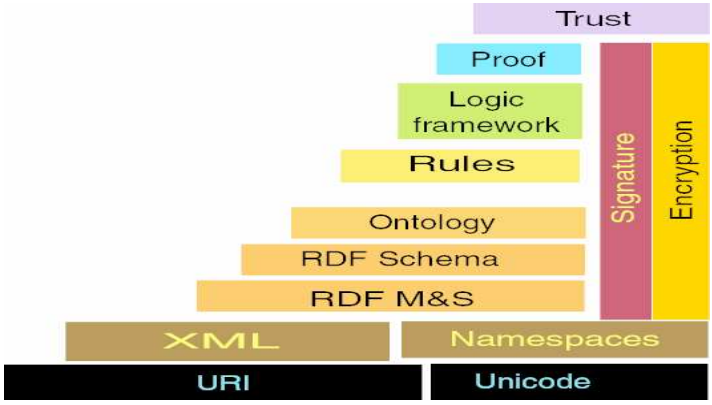


그림 7 시멘틱 웹의 계층 구조

논리기반 지식베이스 시스템에서는 무엇이 진실임을 기술하며 이론적으로 잘 정의된 추론 절차를 통하여 기술된 진실(true fact)들로부터 필요한 결론을 얻어낸다. 응용분야를 기술하기 위하여 형식 언어인 일차 논리어(first order logic language)를 사용한다. 일차 논리어는 그 의미(semantic)가 잘 정의되어 있으며 그로부터 원하는 답을 끌어내는 추론 절차 역시 잘 정의된다. 응용분야는 일차 논리어를 이용하여 사실(fact)들과 규칙(rule)들로 기술된다. 사실이란 응용분야의 객체들에 대하여 이미 드러나 있는 관계를 나타내며, 규칙이란 드러나 있지는 않지만 이들 객체들 사이에 일반적으로 성립하는 성질이나 속성을 나타낸다. 사실들과

규칙들의 집합을 보통 논리 프로그램(Logic program)이라 하며, 논리 프로그램과 그 관련 추론 기능을 다루는 분야를 논리 프로그래밍(Logic Programming)이라 한다. 1970 년대에 들어서서 논리를 프로그램 언어로 사용할 수 있는 연구가 되어 그 결과로 등장한 것이 프롤로그(Prolog)이다. 프롤로그는 일차 논리어의 특수한 형태인 혼 절(Horn clause)을 사용하여 프로그램을 작성하며, 그 추론 방법은 resolution 에 근거하고 있다. 프롤로그와는 유사하나 좀 더 제약이 가해지고 그 의미를 약간 달리하는 언어가 Datalog 이다. Datalog 가 관계 데이터베이스 접근을 위하여 사용되는 경우에는 인터프리트(interpret) 방식의 프롤로그와는 그 추론 절차를 달리하며 특별히 관계 질의어로는 표현이 불가능한 순환 규칙(recursive rule)을 사용한다.

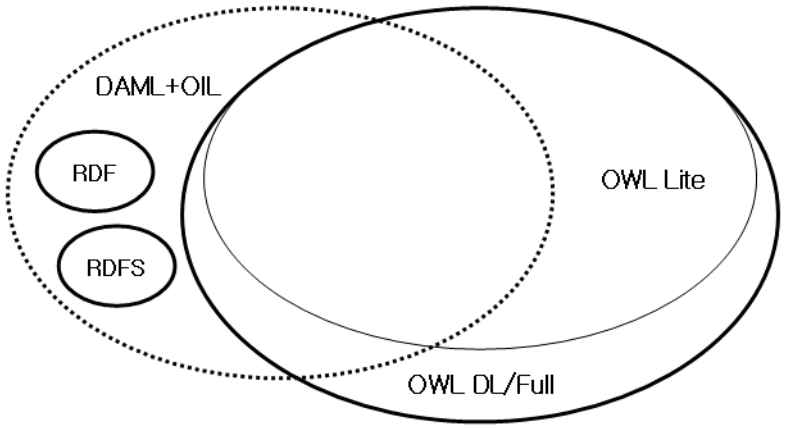


그림 8 온톨로지 언어 어휘간의 포함 관계

일차 논리 언어는 다음과 같이 정의된다. 상수, 변수, 함수 등으로 만들어 질 수 있는 형태를 term 이라 한다. 이 term 들 사이의

관계를 나타내는 것을 Predicate 라 하며 term 을 인수(argument)로 하여 나타낸다. 인수를 갖고 있는 Predicate 를 atom 이라고 한다. 이들 atom 들을 논리 연산자인 and('&'), or('|'), not('~')등과 Universal quantifier('∀')와 Existential quantifier('∃')를 사용하여 연결하면 일차 논리식(First Order Logic Formula)이 된다. atom 또는 atom 에 not 이 붙은 형태를 Literal 이라 한다. not 이 붙지 않은 atom 형태의 것을 Positive Literal 이라 하며 not 이 붙은 것을 Negative Literal 이라 한다. 논리식은 몇 가지 형태로 정규화가 가능하며, 그 중 Conjunctive Normal Form 은 모든 Literal 들이 and 로 연결된 형태를 말한다. Conjunctive Normal Form 이 우리가 나타내려는 실세계를 아무런 문제없이 표현했을 경우(논리 용어로 Consistent 하다고 하며 규칙 상호 간에 모순이 없는 상태를 의미함)에는 그 식의 모든 Quantifier 를 없애고 and 부분에서 모두 잘라 모아 놓은 형태로 변환이 가능하다. 이렇게 잘라진 단위를 절(Clause)이라 하며 각 절은 Literal 들의 ‘or'로만 연결되어 있다. 그러므로 혼 절에 있는 변수는 모두 Universal quantifier 에 의해 제한되며, 절과 절은 ’and'로 연결된 것으로 간주된다. 특별히 그 절 안에 Positive Literal 이 없거나 하나만 있는 형태를 혼 절(Horn clause)이라 부르며, Prolog 와 datalog 는 모두 혼 절만을 사용한다. 정리증명(Theorem proving)이란 절들의 집합을 정리의 가정으로 보고 질의어를 정리의 결론으로 보았을 때 질의어의 답을 끌어내는 과정을 증명(Proof)으로 보는 관점이다. 절들의 집합에 대한 정리증명의 방법으로는 앞서 언급한 resolution 이 보통이다.

resolution 은 이를테면 modus ponens 라는 고전적인 연역 방법의 일반적인 형태로 볼 수 있다. 전체적으로는 질의어를 부정하여 주어진 절들의 집합과 함께 반복적으로 resolution 을 하여 어느 시점에 Empty clause, 즉 모순이 발생하면 증명된 것으로 본다.

3. 관련연구

3.1 에서는 기존의 지식 관리 시스템과 객체 지향 시스템에서 온톨로지와 유사한 계층적 데이터의 처리를 살펴보고, 3.2 에서 Gene Ontology 시스템이 자체적으로 사용하는 트랜지티브 클로저 질의에 대한 처리 기법을 알아 본후, 마지막으로 3.3 에서는 트랜지티브 클로저 질의의 처리에 대해 Jena 시스템에서 사용하는 방식에 대해 기술한다.

3.1 일반적인 지식 관리 시스템에서의 온톨로지

현재까지 온톨로지 관련 연구는 데이터베이스 관점에서 바라보는 대용량을 가정했을 때의 질의 처리의 성능 개선보다는 인공 지능 관점에서의 보다 지능적인 또는 기계가 잘 이해할 수 있는 연구에 초점이 맞추어져 있고, 사실상 그 또한 실험적인 수준에 머물고 있다. 이는 실제적인 응용에서 대용량의 온톨로지 데이터를 찾기 어려운 측면 때문이기도 한데, 그런 의미에서 Gene Ontology[1]는 요즈음 각광 받고 있는 생물학 분야와 결합되어 비교적 대량의 온톨로지 데이터를 제공하고 있다는 점에서 흥미롭다.

온톨로지에 관한 표준으로 RDF/S, OWL 등이 W3C[2]에 의해 제정되어 활발히 연구되고 있으나, 아직까지 실용적인 응용들은 온톨로지의 약한 표현 형태인 택사노미(taxonomy) 정도에 머물고 있고, [3]에서는 ODP 카탈로그를 데이터로 한 거대한 택사노미의

계층 구조 탐색에 대해 세가지 레이블링 기법을 비교 분석하고 있다. Bit-vector 와 prefix, 그리고 구간 기반 레이블링이 그것인데, 결과적으로 prefix 방식과 구간 기반 방식이 온톨로지에 더 효율적임을 주장하고 있다. 그들이 비교한 기법에서 보듯이이 실제로 계층 구조에서의 상속관계에 관한 효율적인 탐색은 기존의 지식관리 시스템에서도 다루어 진 바 있으나[4] 이는 메모리를 효율적으로 이용하기 위한 압축기법에 치중하고 있고, 하나의 상속관계 만을 다루므로 현재의 온톨로지 시스템에 대한 적용에는 미흡한 면이 있다.

[5]는 역인덱스를 사용하여 관계형 데이터베이스를 효율적으로 활용하는데 이는 성능 측정에 대한 아이디어를 제공하고 있다.

[6-8]은 현재의 대표적인 온톨로지 저장소로서 특히 Jena[7]에서는 추론 엔진(reasoner)을 구현하여 제공하고 있으나 그 성능은 미약한 상태이고 특히 트랜지티브 클로저 질의를 처리하는 모듈인 transitive reasoner 는 트랜지티브 리덕션(transitive reduction)을 통해 공간 효율성을 높이고 트랜지티브 클로저에 대해서는 메모리 캐쉬를 하는 단순한 방법을 사용하고 있다.

구체적으로 살펴보면, Jena 시스템은 RDF 그래프에 대한 추상화된 온톨로지 모델을 유지하는데 이때 중간에 Reasoner 에 의해 확장된 그래프를 모델로 가지게 된다. 이 확장은 Jena 의 Reasoner 패키지이고 그 중 transitive reasoning 에 관여하는 것이 transitive reasoner 이다.

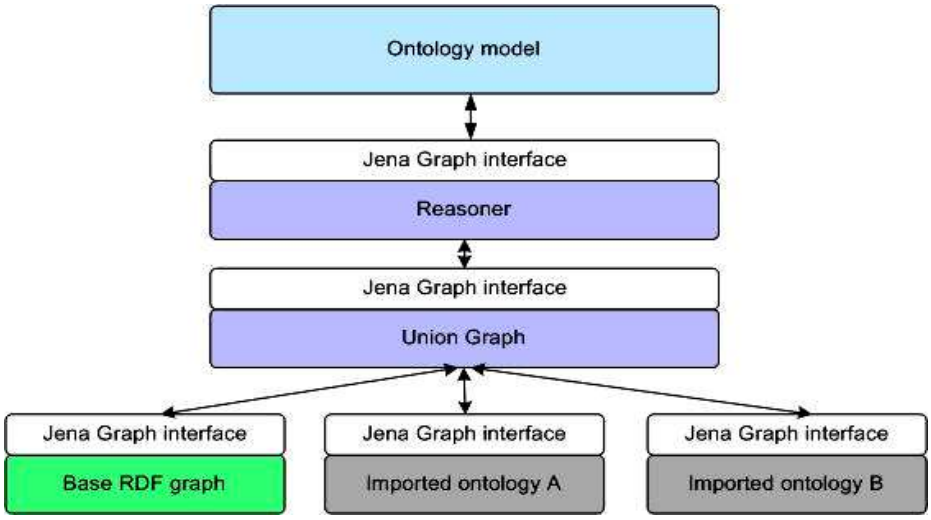


그림 9 Jena 추론 시스템 구조

Jena 에서는 각각의 임포트된 온톨로지 문서가 별개의 그래프 구조에 담기게 되고 만약 이렇게 하지 않으면 한번 임포트된 온톨로지가 어디에서 왔는지 알 수 없게 된다면 점에서 중요하다. 또한 임포트는 재귀적으로 수행 될 수 있는데, 다시 말해서 어떤 문서가 온톨로지 A 를 임포트하고 있고, A 는 B 를 임포트 한다면 결국 그림 15 와 같은 온톨로지 모델을 얻게 된다.

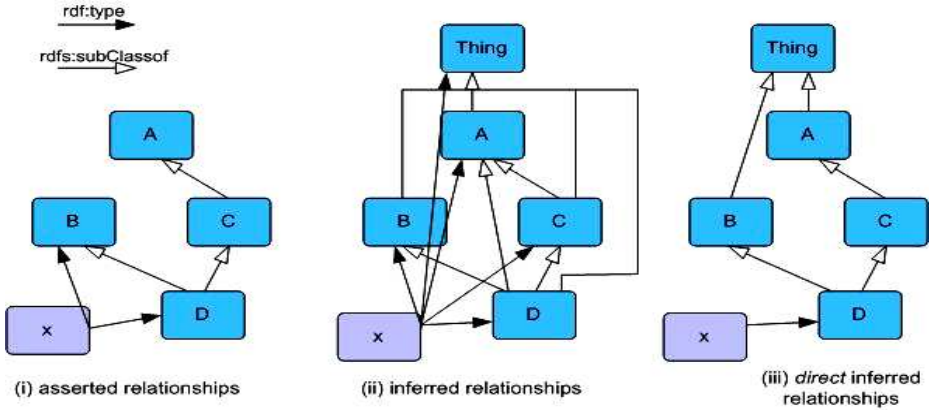


그림 10 Jena 에서의 관계(Relationship) 유형

그림 16 의 i)는 클래스 구조와 인스턴스 X 를 포함하는 기본 모델을 보여주고 있는 반면, ii)는 이 기본 모델에서 유추된 것을 의미한다. Jena 에서는 iii)과 같이 direct inferred relationship 만을 유지 함으로써 저장 공간의 낭비를 막고 있다.

예를 들어 인스턴스 X 가 클래스 C 의 타입을 가진다는 것은 ii)에서 명시적으로 알 수 있지만, iii)에서는 단지 인스턴스 X 가 클래스 D 의 타입을 가지고 클래스 D 는 클래스 C 의 하위 클래스라는 사실로 원래의 의미가 유추 될 수 있는 것이다.

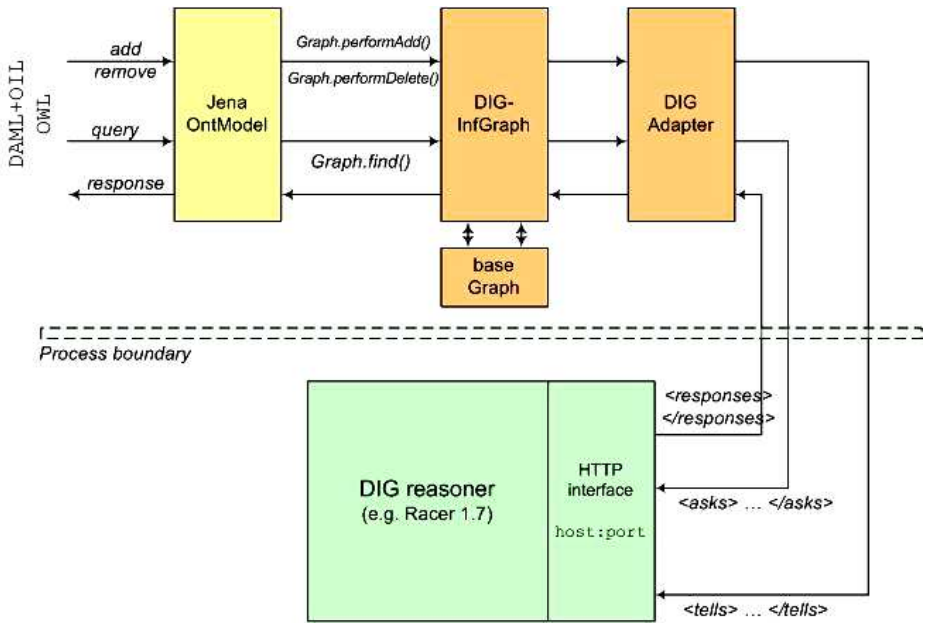


그림 11 Jena 의 DIG 인터페이스

Jena 에 기본적으로 내장된 추론기(reasoner)들은 OWL-lite 와 OWL-DL 그리고 OWL-Full 의 일부를 지원할 수 있으나 대용량의 크고 복잡한 온톨로지에 대해서는 위와 같은 rule-based 접근 방식은 그 복잡도가 매우 크고 느리게 동작한다. 따라서 Jena 가 가지고 있는 내장 추론기(built-in reasoner)외에도 기존의 인공지능 추론 시스템에서 연구되어 온 'Race'나 'Fact', 'Pellet' 와 같은 시스템들을 Jena 에 붙여서 사용할 수 있게 하고, 그러한 인터페이스로 제공되는 것이 DIG 이다. DIG 은 HTTP 기반의 인터페이스로서 점점 DL 에 기반한 추론 엔진을 위한 표준으로 자리 잡아가고 있다.

3.2 GO 시스템에서의 트랜지티브 클로저 질의 처리

Gene Ontology 의 그래프 구조를 DBMS 에 저장하는 구조는 기본적으로 **term** 과 **term2term** 테이블에 의해 유지되고, relationship 종류로는 is_a, part_of, develops_from 이 있다. 온톨로지 질의 처리를 위해서는 그래프 탐색이 필요하게 되고, 이는 **term2term** 테이블에 대한 재귀적인 질의 형태가 되나, 대부분의 SQL 구현들은 이를 지원하지 못하므로 일반적으로 반복적인 SQL 호출의 형태가 되게 된다. 하지만, 이런 방식은 공간에 대한 비용이 대단히 커지므로 Gene Ontology 에서는 임의의 노드에서 갈 수 있는 모든 트랜지티브 하위 클래스를 미리 계산해 놓고(즉, 트랜지티브 클로저), 이를 **graph_path** 테이블에 저장한다. 정확히 말해서 노드 자신도 포함하는 리플렉시브 트랜지티브 클로저(reflexive transitive closure)를 계산한다.

예를 들어, 그림 2 에서 모든 'engyme' 유전자(gene)를 찾으라는 질의에 답하기 위해서는

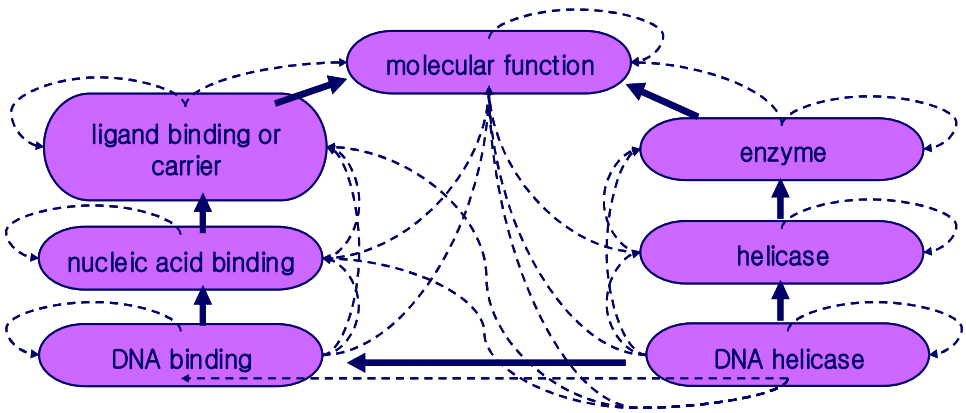


그림 12 Gene Ontology 에서의 트랜지티브 클로저 질의

'engyme' 뿐만 아니라 'helicase' 와 'DNA helicase' 등 engyme 의 transitive closure 에 속하는 모든 노드들의 유전자 또한 답이 된다(총 7 개). Ontology 그래프의 노드가 n 개라고 할 때 그래프 자체는 O(n)에 탐색이 가능하지만, 그것의 트랜지티브 클로저 그래프는 $\sum_{k=1}^{n} k$ = O(n^2)이 되는데 Gene Ontology 시스템에서는 그림 2 의 G^*를 저장하고 있는 셈이므로 대용량의 Ontology 를 가정했을 때 매우 공간 효율적이지 못하고 따라서 효율적인 기법이 요구된다.

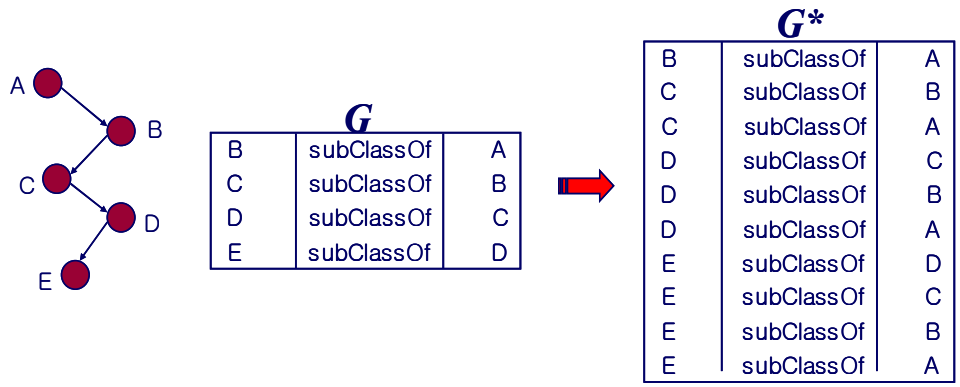


그림 13 트리플 관점에서의 트렌지티브 클로저

3.3 Jena 에서의 트랜지티브 클로저 질의 처리

그림 2 에서 만약 G^* 를 동적으로 계산하려고 한다면 대용량의 온톨로지가 데이터베이스(persistent triple store)에 저장되어 있다고 가정했을 때 계산과정에서 엄청난 양의 I/O 를 필요로 하게 되고 이는 시간 효율성의 저하로 나타난다. 구체적으로는 G 에서 G^*를 구하는 것은 재귀 순환(recursion) 형태의 질의로서 이는 재귀 호출(recursive call)에 해당하는데 대부분의 SQL 구현들은 이를 지원하지 못하므로 여러 반복적인 SQL 문장으로 해결해야 한다.

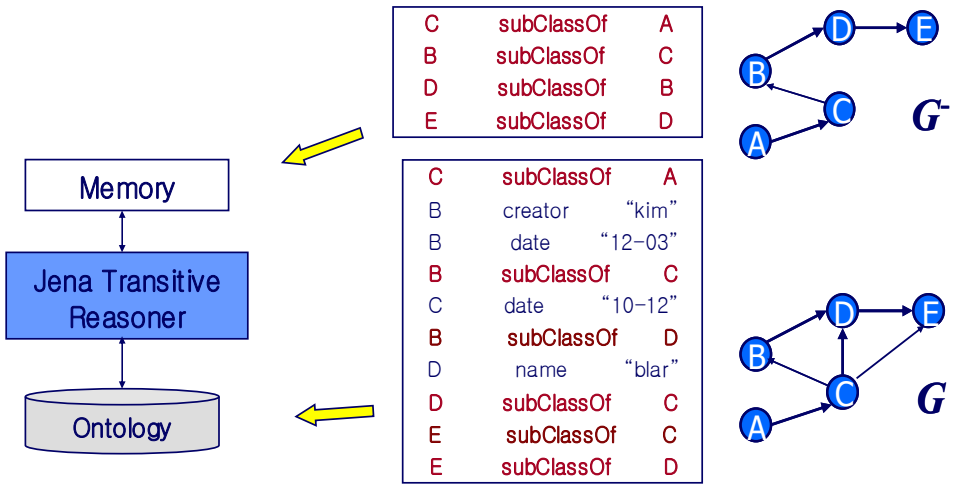


그림 14 Jena 에서의 트랜지티브 클로저 질의 처리 방식

대표적인 온톨로지 저장소인 Jena 에서는 데이터베이스에서 트랜지티브 클로저 계산에 필요한 트리플들을 메모리로 복사해 와서 메모리 상에서 다시 재구성하는 방법을 사용하고 있다. 다시 말해 온톨로지 그래프의 트랜지티브 리덕션(transitive reduction)을

통해 필요한 저장공간을 줄이고 트랜지티브 클로저(transitive closure)에 대해서는 그 그래프에서의 탐색을 통해 갈수 있는 모든 노드를 직접 방문하여 찾고, 이를 메모리에 캐시 하는 정책을 사용한다. 하지만 이는 대용량의 온톨로지를 가정했을 때는 트랜지티브 클로저 계산에 필요한 트리플들이 모두 메모리로 올라올 수 없고, 따라서 필요할 때 마다 디스크 I/O 를 수행해야 하고 이는 큰 부담이 된다.

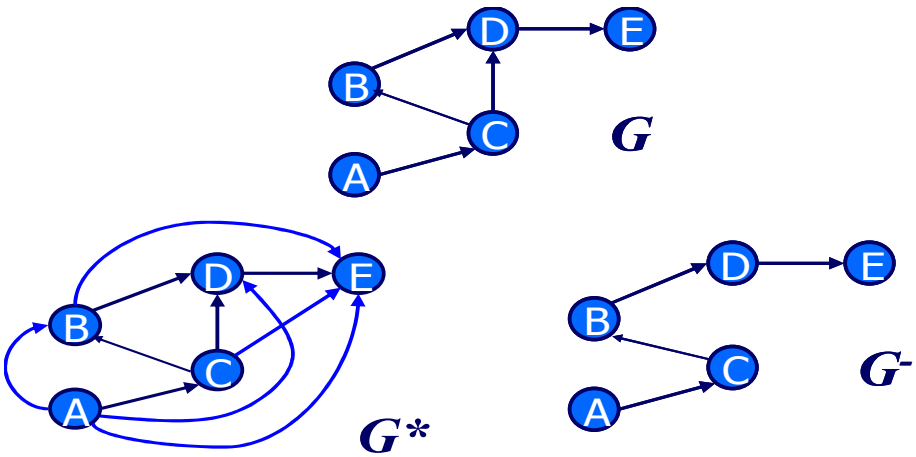
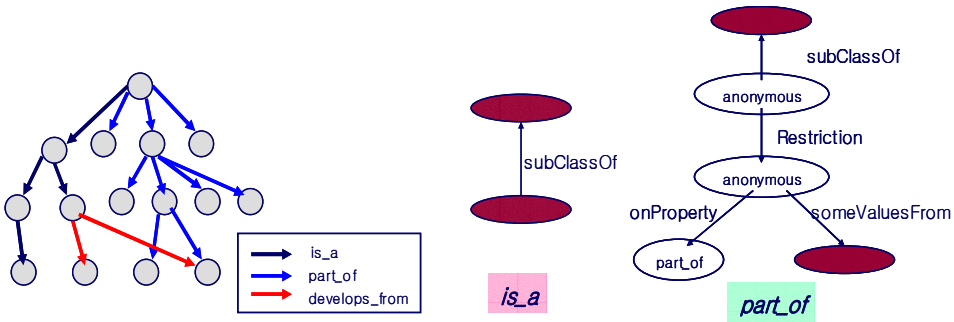


그림 15 트랜지티브 클로저(G^*)와 트랜지티브 리덕션(G^-)

또한 온톨로지에서의 관계는 반드시 한 종류의 상속 관계로 표현될 필요는 없다. Gene Ontology 에서는 현재 is_a 와 part_of, 그리고 develops_from 을 지원하고 있는데 앞으로 더욱 세분화(specialization) 될 예정이라고 한다. W3C 의 표준 온톨로지 언어인 OWL 에서는 subClassOf 와 subPropertyOf 로만 상속관계가 표현되므로 Gene Ontology 의 클래스간의 part_of

관계를 표현하기 위해서는 OWL 의 Restriction 을 이용해야 한다. 즉 Restriction 으로 선언하고 onProperty 에 part_of 관계임을 명시하는 방식인데 이것의 문제점은 기존에 한 개의 트리플로 표현 가능했던 관계가 4 개의 트리플로 표현되게 되고 결국 어떤 클래스의 part_of 관계인 하위 클래스를 찾는 것은 4 개의 트리플을 조인하는 연산을 발생시킨다. 단지 하나의 하위 클래스가 아니라 어떤 클래스의 트랜지티브 클로저를 계산하게 된다면 엄청난 양의 트리플을 더 찾아보게 되는 것이 되므로 이 또한 전체적인 반응시간에 저하를 가져오게 된다. 그림 5 는 실제 Gene Ontology 파일에서 is_a 와 part_of 관계의 트랜지티브 클로저가 트리플 레벨에서 어떻게 찾아지는 지를 보여주고 있다.




```
<owl:Class rdf:about="http://www.geneontology.org/go#stress+fibers">
  <rdfs:subClassOf>
    <owl:Class rdf:about="http://www.geneontology.org/go#part_of_cellular_component"/>
  </rdfs:subClassOf>
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:onProperty rdf:resource="http://www.geneontology.org/go#part-of"/>
        <owl:someValuesFrom>
          <owl:Class rdf:about="http://www.geneontology.org/go#actin+cytoskeleton"/>
        </owl:someValuesFrom>
    </owl:Restriction>
  </rdfs:subClassOf>
</owl:Class>
```

gene ontology file

그림 16 Jena 에서의 is_a 와 part_of 처리 방식 비교

4. 레이블링을 이용한 트랜지티브 클로저 질의 처리 기법

4.1 에서는 기존의 XML 등에서 사용된 구간 기반 레이블링 기법을 그래프로 확장한 기법에 대하여 설명하고 4.2 에서는 그것을 저장하기 위한 자료 구조와 알고리즘을 제안한다. 4.3 에서는 온톨로지의 변경을 가정했을 때의 점진적인 갱신(incremental update)에 대해 기술한다.

4.1 그래프에 대한 구간 기반 레이블링 [10]

그래프가 하나의 연결 컴포넌트(connected component)로 구성되어 있다고 가정한다.
(forest 의 경우는 가상의 노드를 루트로 만들면 동일해짐)
이때 레이블링의 절차는 다음과 같다. 레이블 포맷: (start_number, end_number)

1. 그래프를 깊이-우선 탐색(depth-first search) 하면서 start_number 와 end_number 를 동일한 값으로 할당한다.
2. 자식 노드를 모두 방문하고 부분적인 루트를 방문하게 되면 자식 노드의 start_number 중에 제일 큰 값을 end_number 로 할당한다.
3. 동일한 과정을 전체 그래프의 루트로 돌아올 때 까지 반복한다.

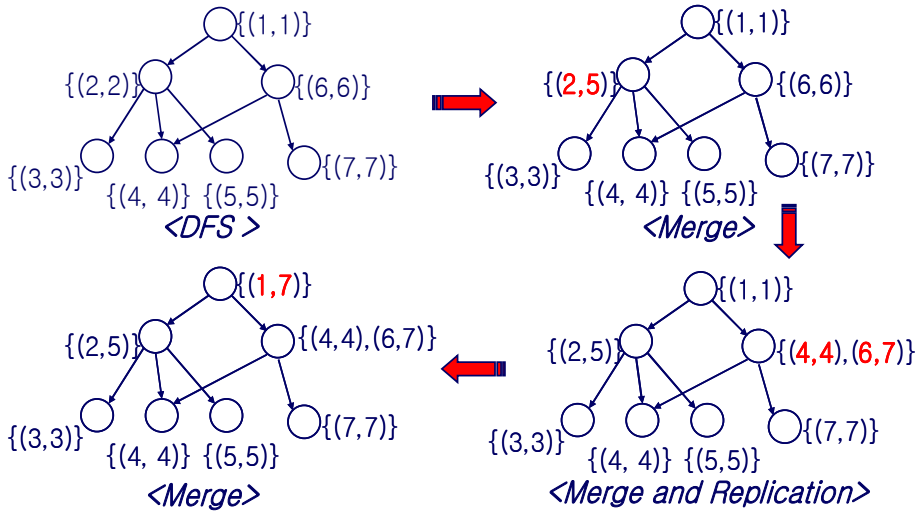


그림 17 구간 기반 레이블링 과정

결과적으로 한 노드를 구별하는 구별자(id)는 그 노드의 레이블에서 start_number 가 된다. 이때, 중요한 것은 트리 간선(tree edge)이 아닌 그래프 간선(graph edge)에 대해서는 해당 자식 노드의 레이블 값을 추가하게 된다. 따라서 트리와 달리 일반적인 그래프에서는 부모가 여러 개인 노드가 있을 수 있으므로 노드의 레이블은 구간(interval)들의 집합으로 표현되고, 전체 레이블이 차지하는 공간은 평균적으로 원래 그래프와 같은 O(n) 임을 알 수 있다. 최악의 경우로 이등분할 그래프(bipartite graph)에서의 레이블링은 $O(n^2)$이 될 수 있으나, 일반적인 온톨로지에서는 이런 경우를 가정하기 힘들다. 이러한 레이블링이 주는 장점은 그래프 탐색을 하지 않고도 레이블 정보만 보고 두 노드 사이의 연결 가능성(Reachability)을 결정할 수 있다는 점이고,

이는 원래 $O(n^2)$ 인 트랜지티브 클로저 그래프를 O(n)으로 압축하여 표현한 효과를 가져온다. 예를 들어 루트 노드 (1,7)과 단말 노드 (5,5) 는 루트 노드의 구간이 단말 노드의 구간을 포함하므로 루트 노드에서 단말 노드로 가는 경로가 존재함을 의미하게 된다. 온톨로지 에서의 트랜지티브 클로저 질의는 단순한 접근 가능성 결정 보다는 한 노드에서 갈 수 있는 모든 노드를 구하는 것이 주가 되는데, 이는 적절한 자료구조를 이용하여 탐색범위를 최소화 할 수 있으며, 4.2 절에서 설명한다.

4.2 자료 구조

트랜지티브 클로저에 대한 탐색은 구간들에 대한 리스트에 대해 start_number 에 B^+-tree 인덱스를 생성한 후 효과적으로 찾을 수 있다. 이때 리스트의 원소는 구간과 해당 노드의 URI 를 쌍으로 갖고, 이에 대한 구현은 관계형 데이터 베이스를 이용할 수도 있고, 직접 인덱스를 구현할 수도 있다.

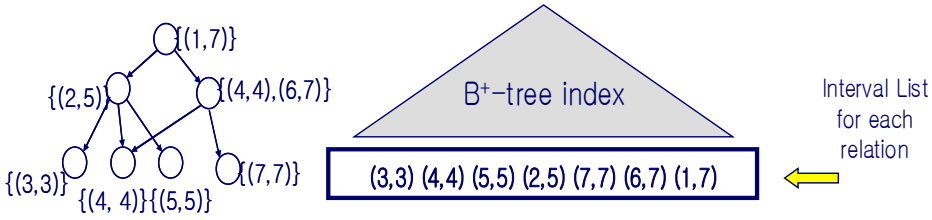


그림 18 레이블링된 데이터에 대한 저장 자료 구조

예를 들어 그림 7 에서 (1,7)에 해당하는 노드의 모든 트랜지티브 하위 클래스를 찾기 위해서는 이미 레이블링 정보가 저장되어 있는 리스트를 탐색하면서 (2,5)에 포함되는 구간에 해당하는 URI 를 찾으면 답이 된다. 이때 리스트가 start_number 로 정렬되어 있다는 조건을 이용하여 start_number >= 2 인 노드만을 탐색하면서 end_number<= 5 인 노드를 찾기 때문에 전체 리스트에 대한 탐색보다 효율적이다.

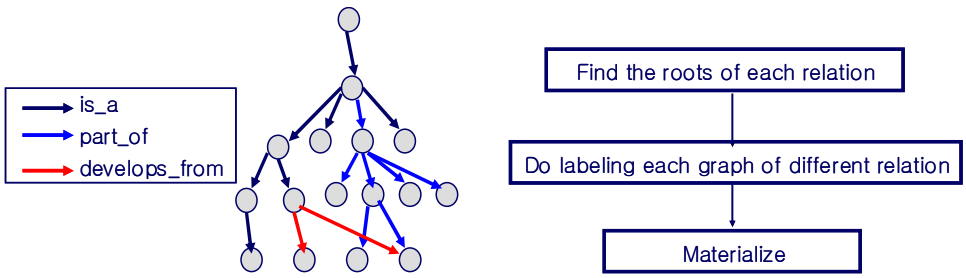


그림 19 온톨로지의 여러 관계에 대한 레이블링 고려

또한 Gene Ontology 의 예에서 보듯, 온톨로지 에서의 트랜지티브 관계(transitive relationship)은 is_a 관계 이외에도 part_of 나 develops_from 등 좀 더 세분화 된 관계들이 존재 할 수 있는데, 2.3 절에서 언급한 바와 같이 이러한 모든 관계가 OWL 스펙에서 subClassOf 로 표현 되기 위해서는 Restriction 을 사용해야 하고, 이는 트리플 기반의 온톨로지 저장소에서 더 많은 트리플 탐색의 비용을 가져 온다.

따라서 레이블링을 하는 전처리 과정에서 온톨로지 내에 존재하는 모든 트랜지티브 릴레이션에 대해 각각 리스트를 유지하고 질의 처리시에 이용한다면, is_a 관계와 똑 같은 비용으로 part_of 에 대한 트랜지티브 클로저를 찾을 수 있다.

4.3 알고리즘

<정의 1> 이항 관계(Binary Relation)
두 집합 A, B 에 대하여 A 로부터 B 로의 이항 관계 R 은 두 집합의 곱집합 A x B 의 부분 집합이다. A x B 의 원소인 순서쌍 (a, b)가 주어졌을 때, a R b 는 (a, b) ∈ R 의 필요 충분 조건이다.

<정의 2> 리플렉시브 관계(Reflexive Relation)

집합 A 에 있는 모든 원소 x 에 대하여 x R x 이면, (x, x)∈ R 이면 관계 R 을 리플렉시브 관계라고 한다.

<정의 3> 트랜지티브 관계 (Transitive Relation)

집합 A 에 있는 원소 x, y, z 에 대하여 관계 R 이 $(x, y) \in R$ 이고 $(y, z) \in R$ 이면 $(x, z) \in R$ 인 관계를 만족하면 관계 R 을 트랜지티브 관계라고 한다.

<알고리즘 1> 리플렉시브 트랜지티브 클로저 알고리즘

```
listSubclasses(target)
{
      for i = target.start to target.end
            find node of I
            add to result
      return result
}
```

```
listSupersubclasses(target)
{
      for each node s.t. node.end >= target.end
         if node.start <= target.start
            add to result
      return result
}
```

```
getNCA(target1, target2)
{
         let target1 to have larger postorder number
         for each node s.t. node.end >= target1.end
            if node.start <= target1.start and
                node.start <= target2.start
               return node
}
```

4.4 점진적인 갱신 (Incremental update)

온톨로지는 사실상 스키마의 확장이고 관계형 모델에서의 스키마는 거의 변하지 않는다고 가정할 수 있지만, 온톨로지의 경우는 비교적 갱신이 빈번히 일어날 수 있다. 예를 들어 야후나 ODP 카탈로그의 경우만 보다라도 필요에 의해 새로운 분류항목이 언제든지 추가 될 수 있어야 하고, 이때 4.1 절의 레이블링을 그때마다 다시 하는 것은 효율적이지 못하므로 점진적인 갱신(incremental update)이 필요하다. 사실 start_number 를 반드시 연속적인 정수로 줄 필요는 없다. 즉 적당한 간격(gap)을

두어 할당하면 전체적으로 레이블링을 새로 하지 않고 새로운 노드에만 비어 있는 레이블을 할당할 수 있게 된다. 갱신을 삽입(insert)과 삭제(delete)의 경우로 나누어 생각해 보면, 삭제(elete)는 해당 노드를 그냥 지우면 되고, 삽입의 경우는 비어 있는 start_number 값을 사용하면 해결된다. 그림 10 은 gap = 10 일 때의 노드의 삽입을 보여주고 있다.

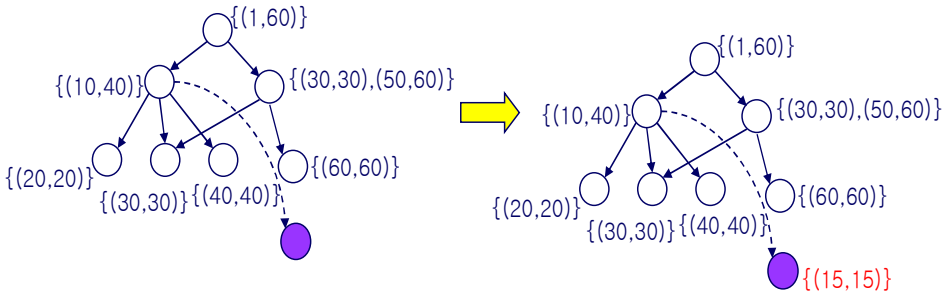


그림 20 새로운 노드가 삽입 될 때

삽입과 삭제를 고려하는 경우에는 어떤 클래스의 모든 하위 클래스를 구할 때 레이블 구간에 해당하는 그 하위 클래스들이 존재하지 않는 경우도 발생하게 된다. 예를 들어 그림 10 에서 (10,40)의 하위 클래스는 노드 10 부터 40 까지에 해당하지만 실제로는 4 개밖에 존재하지 않는다. 하지만 노드 ID 에 대한 실제 URI 는 해쉬 테이블에 저장되어 있고, 이런 경우는 해쉬 테이블에 해당 키값이 없는 경우이므로 상수 시간에 모든 하위 클래스를 찾을 수 있는 것에는 변함이 없다.

4.5 이론적 분석

4.5.1 공간 효율성

Gene Ontology 시스템처럼 모든 트랜지티브 클로저 그래프의 간선을 미리 계산해서 저장한다면 $\sum_{k=1}^{n} k$ = O(n^2) 이 된다. Jena 의 경우 그래프에 대한 트랜지티브 리덕션(transitive reduction)을 구하지만, 복잡한 네트웍과는 달리 온톨로지는 이로 인해 제거되는 간선의 수는 많지 않고 결국 O(n)의 공간 복잡도(space complexity)를 갖는다. 구간 기반 레이블링을 이용한 우리의 접근 방법도 노드 당 평균적으로 1 개의 레이블을 유지하므로 O(n)의 공간 복잡도를 갖는다. 여기서 n 은 트리플의 수 또는 노드의 수를 말한다.

4.5.2 시간 효율성

한 클래스의 모든 서브 클래스를 찾는 질의를 생각했을 때 메모리에서의 처리를 가정한다면 Jena 의 경우 O(n)이 되는데 이는 답이 되는 노드를 찾기 위해서는 그 노드를 직접 방문해서 찾기 때문이다. 반면에 구간 기반 레이블링을 이용한 우리의 접근 방식은 그 클래스의 레이블만 보면 마치 해쉬 함수 처럼 답이 되는 클래스를 찾을 수가 있으므로 O(1)이 된다.

그림 11 에서 Jena 의 경우는 DFS 로 재귀적으로 모든 해당 노드를 탐색하게 되고 이는 결국 답이 되는 노드의 수가 n 이라고 할 때

O(n)의 시간 복잡도를 갖게 된다. 구간 기반 레이블링을 이용한 우리의 접근 방법은 한 노드의 레이블이 여러 구간이 될 수 있으나 결국 상수이고, 따라서 수행 시간도 O(1)의 시간 복잡도를 갖게 된다.

하지만, 모든 상위 클래스를 구하는 질의는 레이블만 보고 결정될 수 없기 때문에 이론적으로 Jena 와 같은 O(n)이고, 이는 NCA 의 경우도 마찬가지다. 하지만 트랜지티브 클로저 질의는 트랜지티브하게 갈 수 있는 모든 하위 클래스를 구하는 것이고, 상위 클래스에는 해당하지 않는다. 또한 현재 Jena 는 메모리에서의 처리를 가정하고 있으므로, 만약 트랜지티브 클로저 질의 처리에 필요한 트리플들을 모두 메모리로 올릴 수 없을 정도의 대용량의 온톨로지를 가정한다면, 결국 Jena 의 방식은 Gene Ontology 시스템의 방식과 큰 차이가 없게 된다.

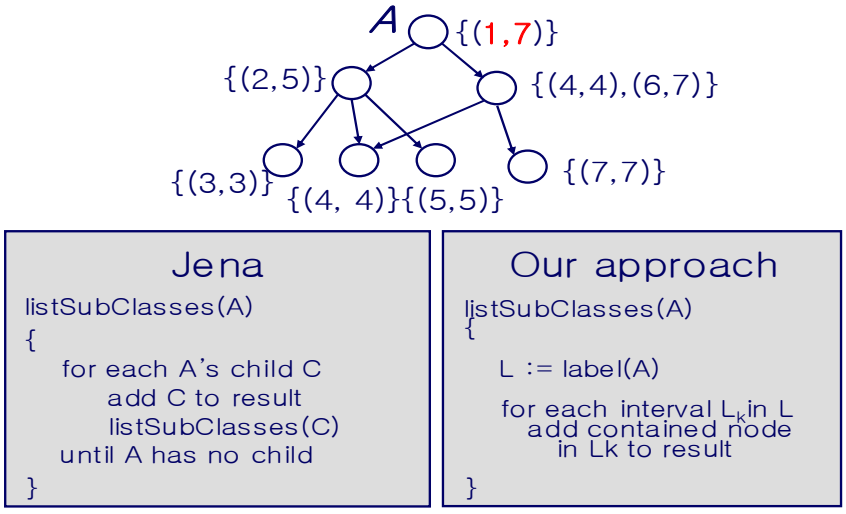


그림 21 트랜지티브 하위 클래스를 찾는 알고리즘

5. 실 험

5.1 실험 환경

실험은 Pentium Ⅲ 1.1Ghz CPU, 512MB 메모리, Windows XP 운영체제, 그리고 Jena2.1 과 MySQL 4.0.20 에서 수행하였다. 실험에 쓰인 데이터는 Gene Ontology 의 term-db 로서 이는 gene_product 정보를 포함하고 있는 association-db 와는 달리 순수한 스키마 정보만을 가지고 있다. GONG[13] 프로젝트에서 제작한 DAML 포맷의 파일을 OWL 로 변환한 후 실험 데이터로 사용하였다. 전체적으로 14000 여개 정도의 클래스를 가지고 최대 깊이(depth) 13 의 구조를 가지고 있는데 한 노드에서 나가는 간선의 수(fan-out)가 상당이 크고 여러 부모를 갖는 노드(term)들이 많기 때문에 트랜지티브 클로저 질의에 대한 실험용으로 적당하다고 볼 수 있다. 관계(relationship)는 19000 여개 로서, is_a 가 17000 여개 이고 part_of 가 2000 여개 이다. (표 2)

실험 방법은 세가지 방식을 네 가지 질의에 대해 비교하고, 측정 결과는 다섯 번의 평균치를 millisecond 단위로 구하였다. 여기에서 세가지 방식은 naïve 방식과 Jena 의 방식, 그리고 구간 기반 레이블링을 이용한 방식을 의미하고, 네 가지 질의는 표 3 에서 보는 바와 같이 subclass (is_a), subclass(part_of), superclass, Nearest Common Ancestor(NCA)에 대해 수행하였다.

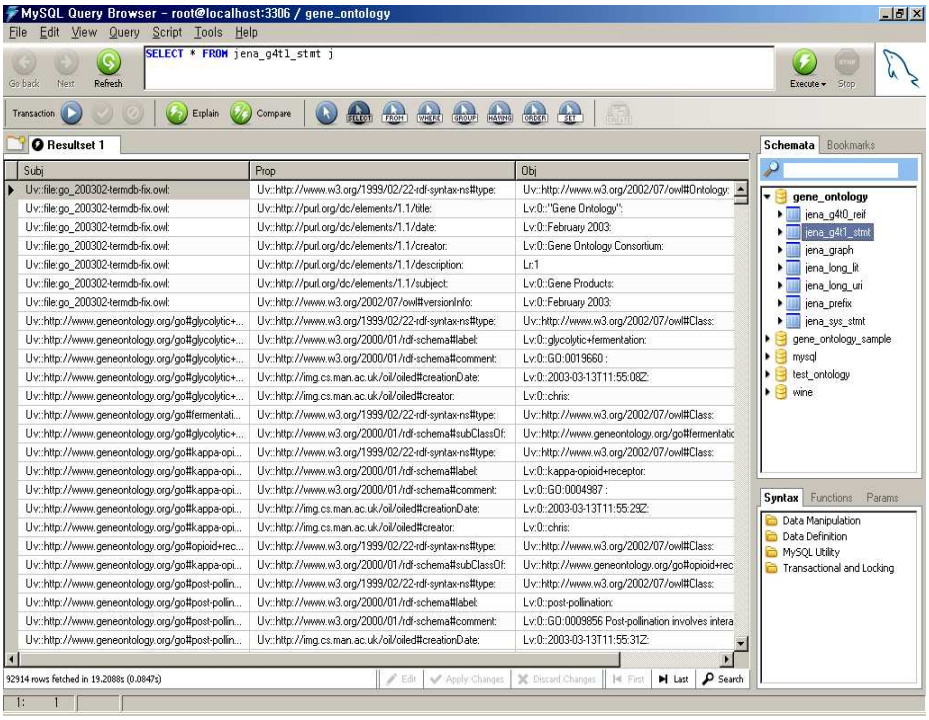


그림 22 MySQL에 저장되어 있는 트리플 데이터

표 2 실험에 사용된 Gene Ontology 스펙

	Molecular function	Biological process	Cellular component	Total
Term	5399	7309	1304	14012
Edge	6856	11202	1644	19702

* is_a: 17602, part_of: 2100, total: 19702

표 3 실험에 사용된 테스트 질의

Q1	Find all (is_a) subclasses of one class
Q2	Find all (part_of) subclasses of one class
Q3	Find all superclasses of one class
Q4	Find the nearest common ancestor of two classes

5.2 실험 결과

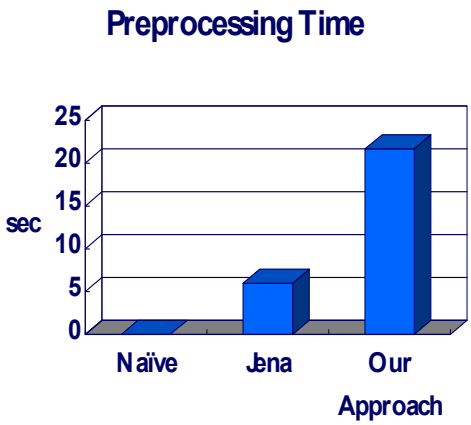


그림 23 기법별 전처리 시간 결과

그림 24 에서 전처리 시간을 비교해 보면, naïve 방식은 전처리 시간이 필요 없고 Jena 에 비해서 본 논문의 방식이 4 배 정도의 시간을 필요로 한다. 이는 Jena 의 단순한 트랜지티브 리덕션 전처리에 비해서 그래프 레이블링이 복잡한 것에 기인하고, 더더욱 트리 레이블링에 비해서 그래프 레이블링은 병합하는 한 단계를 더 거치게 된다. 따라서 대용량 온톨로지를 나타내는 그래프에서는

비교적 많은 시간을 필요로 하는 것이 사실이지만, 이는 충분히 가능한 시간이고 한번 수행하면 그 다음부터는 반복적으로 질의에 답할 수 있고 전처리 시간은 문제가 되지 않는다.

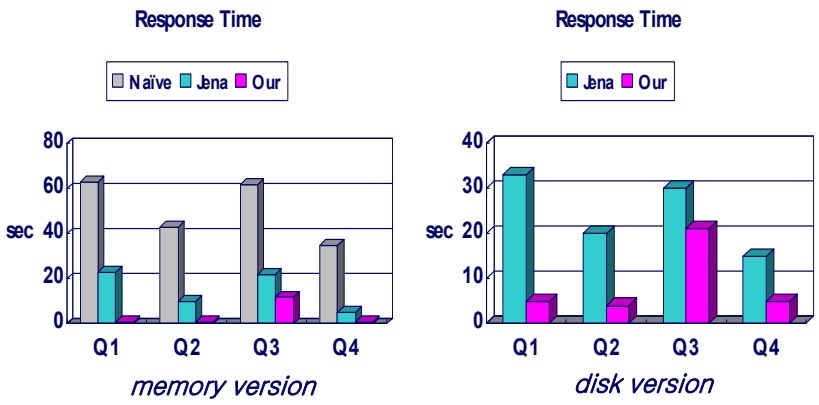


그림 24 질의에 대한 응답 시간 결과

응답 시간에 대한 실험은 Jena 가 메모리 처리 방식 위주이므로 그것을 naive 방식처럼 수정해서 디스크 기반의 실험을 추가하였다. 그림 25 의 첫 번째 차트는 메모리 버전에 대한 결과이고 두 번째 차트는 Jena 를 naïve 방식처럼 수정하고 본 논문에서 제시하는 기법을 디스크 기반으로 했을 때의 응답 시간에 대한 결과이다.

naïve 방식은 많은 SQL 문장으로 인해 디스크 I/O 가 많이 발생하므로 규모가 큰 ontology 에 대해서는 거의 불가능해 보이는 반면 Jena 나 우리의 방식은 비교적 작은 응답 시간을 보이고 있는데 특히 엄밀한 의미의 트랜지티브 클로저 질의에 해당하는 Q1 과 Q2 에 대해 우리의 방식이 훨씬 작은 응답 시간을 보이고

있다. Q3 는 모든 상위 클래스를 구하는 질의에 해당하고 이는 4.4 의 시간 복잡도 분석에서 보듯이 본 논문에서 제시하는 기법의 경우에도 답이 되는 모든 노드들을 리스트에서 찾아야 하므로 응답 시간에 있어서 Jena 의 방식과 큰 차이를 보이지 않고, Q4 의 NCA 를 찾는 질의는 Q3 의 특수한 경우라고 볼 수 있다.

6. 결 론

본 논문에서는 구간 기반 레이블링을 이용하여 그래프 기반의 데이터인 온톨로지 에서의 트랜지티브 클로저 질의에 대한 효율적인 처리와 여러 트랜지티브 관계를 고려한 처리에 대한 방법을 제안하였다. 우리의 접근 방법은 구간 기반 레이블링을 이용한 트랜지티브 클로저 그래프 압축과 실체화를 통해 대용량의 온톨로지에 대한 상당한 크기의 트랜지티브 클로저 그래프 정보를 적은 I/O 로 메모리 내에서 처리할 수 있게 한다. Gene Ontology 는 대용량의 데이터와 복잡하고 다양한 관계(relationship)로 인해 좋은 실험 환경을 제공하고, 본 논문이 제안한 기법은 빈번한 트랜지티브 클로저 질의들에 대한 성능을 개선시킬 수 있음을 대표적인 온톨로지 저장소인 Jena 의 방식과 비교하여 이론적으로 그리고 실험적으로 보이고 있다. 또한 실체화된 자료의 갱신은 온톨로지의 갱신되는 정도를 고려해 처음에 설정된 간격(gap) 에 의해 점진적인 갱신(incremental update)이 쉽게 가능하다.

이는 온톨로지의 추론기능 중에 가장 기본적인 트랜지티브 추론(transitive reasoning)에 대한 내용으로 비록 W3C 의 표준 온톨로지 언어인 OWL 의 기능의 일부에 대한 개선에 해당하지만, 트랜지티브 클로즈 질의 자체는 빈번하게 발생하면서 그 비용이 크다는데 개선의 의의가 있고, 앞으로 다양하고 OWL 의 기능을 풍부하게 사용하는 도메인이 많아지더라도 근본적인 요소(building block)로서 상위 시스템에 결합될 수 있을 것이다.

7. 참고문헌

[1] Gene Ontology (GO): http://www.geneontology.org

[2] W3 Consortium (W3C): http://www.w3c.org

[3] V. Christophides, G. Karvounarakis, D. Plexousakis. "Optimizing Taxonomic Semantic Web Quries using Labeling Schemes", *Web Semantics: Science, Services, and Agents on the World Wide Web*, Vol. 1(2), 2004: 207-228

[4] R. Agrawal, A. Borgida, H. V. Jagadish. "Efficient Management of Transitive Relationships in Large Data and Knowledge Bases", *Proceedings of the ACM SIGMOD International Conference on the Management of Data*, 1989:253-262.

[5] C. Zhang, J. Naughton, D. DeWitt, Q. Luo, G. Lohman. "On Supporting Containment Queries in Relational Database Management Systems", *Proceedings of the ACM SIGMOD International Conference on the Management of Data*, 2001:425-436.

[6] S. Alexaki, V. Christophides, G. Karvounarakis, D. Plexousakis. "The ICS-FORTH RDFSuite: Managing Voluminous RDF Description Bases", *Semantic Web Workshop* 2001.

[7] K. Wilkinson, C. Sayers, H. Kuno, D. Reynolds. "Efficient

RDF Storage and Retrieval in Jena2", *SWDB* 2003.

[8] J. Broekstra, A. Kampman, F.V. Harmelen. „Sesame: An Architecture for Storing and Querying RDF Data and Schema Information", Semantics for the WWW, 2001

[9] ODP Project: http://www.dmoz.org

[10]J. Kim, H.-J. Kim, "Efficient Processing of Regular Path Joins using PID", Information and Software Technology, 2002

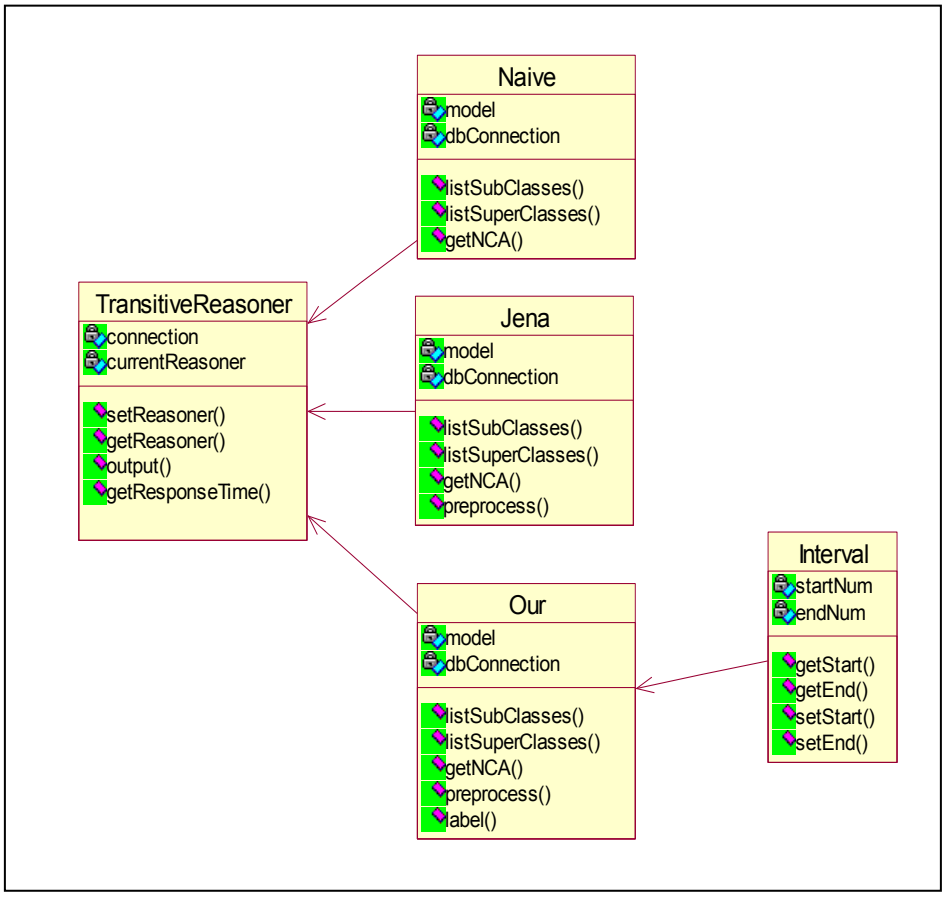
[11] I. Horrocks, S. Tobies, "Optimisation of Terminological Reasoning", Proceedings of International Workshop in Description Logics 2000

[12] G. Karvounarakis, S. Alexaki, V. Christophides, D. Plexousakis, "RQL: A Declarative Query Language for RDF", Proceeding of International World Wide Web Conference 2002.

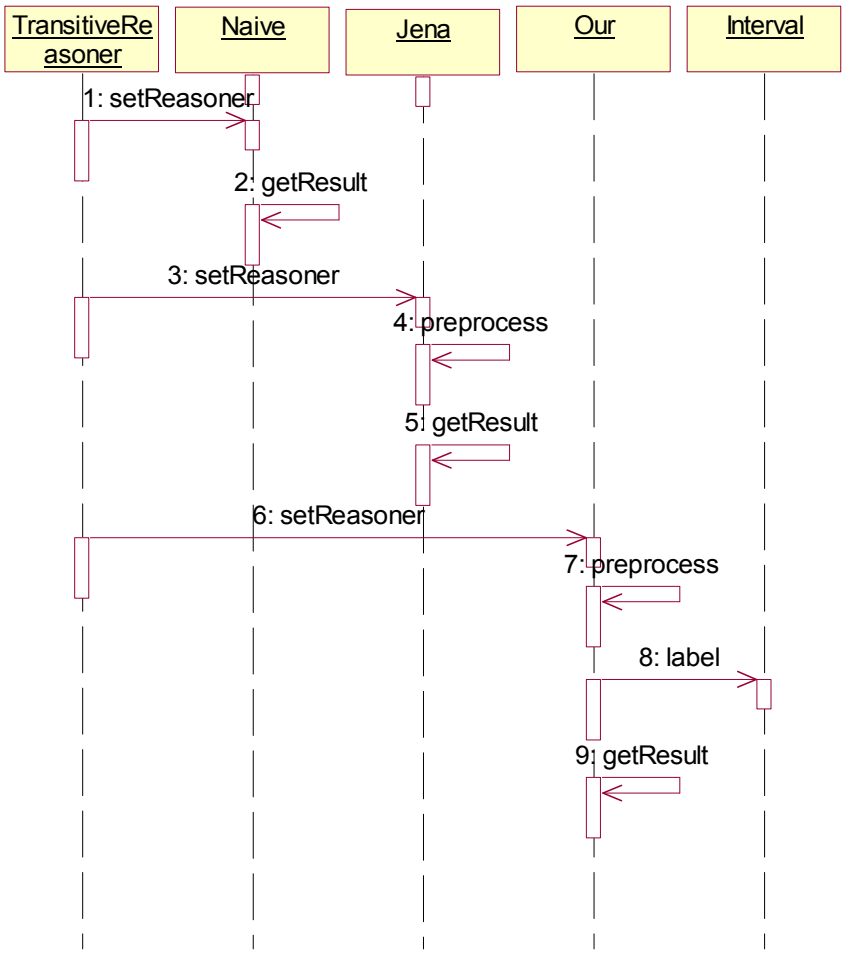
[13] Gene Ontology Next Generation (GONG) Project: http://gong.man.ac.uk/

[14] I. Horrocks, "Reasoning with Expressive Description Logics", Theory and Practice Proceedings of DADE-02, 2002. Springer-Verlab Lecture Notes in Artificial Intelligence.

Class Diagram



Sequence Diagram



Preprocessing Routine

```
protected void preProcess()
{
    try
    {
        String className = "com.mysql.jdbc.Driver";
        Class.forName(className);

        // Create database connection
        IDBConnection conn = new DBConnection ( DB_URL, DB_USER,
    DB_PASSWD, DB );
        Model m = ModelRDB.open(conn, GENE_ONT);

        Resource root =
                m.getResource("http://www.geneontology.org/go#part_of_biologi
          cal_process");
            label(m, root);

        for (Iterator iter = map.entrySet().iterator(); iter.hasNext(); ) {
            Map.Entry entry = (Map.Entry)iter.next();
            LinkedList tags = (LinkedList)entry.getValue();
        }
    } catch (Exception e) {
        e.printStackTrace();
        System.exit(1);
    }
}
```

Labeling Routine

```
protected LinkedList label(Model m, Resource root)
{
      if (map.containsKey(root.getURI())) { // visited
            LinkedList t = (LinkedList)map.get(root.getURI());
            t.add(new Interval(-1,-1));
            return t;
      }
      if (!hasChild(m, root)) {
            LinkedList t1 = new LinkedList();
            Interval v = new Interval(++p, p);
            t1.add(v); map.put(root.getURI(), t1); map2.put(t1, root.getURI());
            return t1;
      } else {
            LinkedList labelSet = new LinkedList();
                  LinkedList minList = new LinkedList();
            Selector s = new SimpleSelector(null);
            LinkedList child = null;
            for (StmtIterator i = m.listStatements(s); i.hasNext(); ) {
                  Statement st =   (Statement)i.next();
                  child = label(m, st.getSubject());
                  Interval t = (Interval)child.getLast();
                        if (t.equals(new Interval(-1,-1))) {
                              child.removeLast();
                              labelSet.addAll(child);
                              minList.addAll(child);
                        } else {
                              minList.addAll(child);
                              map2.put(t, root.getURI());
                        }
            }
            Interval u = new Interval(getMinInterval(minList).getStart(), ++p);
            labelSet.add(u);
            map.put(root.getURI(), labelSet);
            map2.put(labelSet, root.getURI());

                  return labelSet;
            }
      }
}
```

Abstract

Ontology is a methodology to describe specific concepts and the relationship between them. Ontology is useful in semantic web and knowledge representation in that it uses relationship between concepts to represent some concrete semantics of specific concept. When we want to get some information from ontology, we severely have to process transitive relationship because most of relations between concepts represent transitivity. Furthermore, it causes recursive calls to process such transitive closure queries, and the cost is quite heavy.

This paper describes the efficient technique for processing transitive closure queries in ontology store. To the purpose of it, we examine some approaches of current system for transitive closure queries, and propose a technique using graph labeling scheme. Assuming quite larage ontology, we would show that this approach give space and time efficiency analytically and empirically.

Keywords: Ontology, Transitive relationship, Transitive closure, labeling