논문 2010-47CI-4-7

프로그램 분석을 통한 RDF 질의 최적화 기법

(RDF Query Optimization Technique based on Program Analysis)

최 낙 민*, 조 은 선**

(Nak-Min Choi and Eun-Sun Cho)

요 약

시맨틱 웹 프로그래밍은 아직 발전 과정 상 초기 단계로서 API에 의존하고 있어, 컴파일 시 에러 검출이 어려우며 프로그래밍 생산성이나 실행 효율성이 낮다. 이를 극복하기 위한 여러 연구 중 하나로 기존 프로그래밍 언어를 확장하여 시맨틱 웹 데이터 처리를 위한 전용 프로그래밍 언어를 만드는 작업들이 진행되어 왔다. 본 연구에서는 RDF (Resource Description Framework) 전용 프로그래밍 언어인 Jey로 작성된 프로그램의 효율성을 높이기 위한 방법으로 Jey의 SPARQL 지원 구조에 캐싱 기법을 추가하여 성능을 높이는 방법을 제안한다. 프로그램 정적 분석을 바탕으로 캐시 대상이 판별되므로 정확도를 높이며 성능향상에 기여하게 된다.

Abstract

Semantic Web programming is such an immature area that it is yet based on API calls, and does not provide high productivity in compiler time and sufficient efficiency in runtime. To get over this limitation, some efforts have been devoted on dedicated programming languages for Semantic Web. In this paper, we introduce a sophisticated cashing technique to enhance the runtime efficiency of RDF (Resource Description Framework) processing programs with SPARQL queries. We use static program analysis on those programs to determine what to be cashed, so as to decrease the cash miss ratio. Our method is implemented on programs in 'Jey' language, which is one of the programming languages devised for RDF data processing.

Keywords: RDF, Semantic Web, Script Languages, Path Expression, Query optimization

I. 서 론

시맨틱 웹은 인터넷과 같은 분산 환경에서 리소스에 대한 정보를 온톨로지 형태로 표현하고 처리할 수 있게 해준다[1]. 이러한 시맨틱 웹 온톨로지를 기술하는 표준 언어로는 RDF, OWL 등이 있으며, 이를 이용하여 웹상의 정보를 효과적으로 처리할 수 있다.

하지만 이러한 온톨로지를 기술하기 위한 시맨틱 웹 표준 언어들은 작성이 어렵고 에러를 발견하기 어려우며, 문자열 형식으로 서술 및 처리되는 Semantic Web 구문의 오탈자를 찾기 힘들다는 단점이 존재한다[2]. 이러한 단점을 보완하기 위하여 기존의 Java/C++등의 고급 프로그래밍 언어들은 라이브러리 형태로 온톨로지 언어들의 작성 방법을 제공을 하고 있지만[3~4] 문법이 까다롭고 코드의 직관성이 떨어지며, 여전히 사전 에러 검출이 어려워 응용 개발이 복잡해진다는 약점이 있다. 또한 응용 프로그램의 내용을 모르는 상태에서 API가 작성되므로 수행 효율성도 떨어지게 된다.

이를 보완하기 위해 여러 학회에서 스크립트 언어(script language)나 스프레드 시트(spread sheet), 마크업 언어(mark-up language) 등에서 온톨로지를 접근하여 개발 효율성을 높이는 실험적인 연구들이 진행되고

* 학생회원, ** 정회원, 충남대학교 컴퓨터공학과
(Dept. of Computer Science and Engineering, Chungnam National University)

※ 이 논문은 2008년도 정부(교육과학기술부)의 재원으로 한국연구재단의 지원을 받아 수행된 기초연구사업임. (No. KRF-2008-531-D00034)
접수일자: 2010년6월1일, 수정완료일: 2010년7월7일

있다[5]. 이 중 스크립트 언어 등 프로그래밍 언어에 기반을 둔 방법은 일반적인 프로그래밍 언어의 표현력을 유지함으로써 기존의 API기반 방법으로 시맨틱 웹 데이터를 처리하던 프로그래머들에게 친숙한 방법을 제공하면서도, 생산성을 향상시킬 수 있다[6]. 또한, 이러한 언어들로 작성된 프로그램은 프로그램의 정적 분석에 사용되던 기법들을 적용할 수가 있으므로, 이를 통한 성능 향상도 가능하다[7].

그러나 기존의 이러한 시맨틱 웹 전용 프로그래밍 언어들은 이러한 장점을 아직 극대화하지 못하고 있다. 특히, 전용 프로그램으로 작성된 프로그램 내용을 사전에 분석하여 실행 효율을 높이는 방법에 대한 연구는 전무한 실정이다.

본 논문에서는 RDF(Resource Description Framework) 처리를 위한 스크립트 언어 중 하나인 Jey[6]로 작성된 프로그램에 대하여, 질의 내용에 대한 정보를 통해 질의 결과 간의 포함관계를 파악하는 프로그램 분석 기법과 이에 기반을 둔 캐시 방법을 제안한다. 질의 결과 간의 포함 관계에 기반을 두어 질의 결과를 캐싱하는 기법은 특히, 주로 웹상에 분산된 RDF 데이터베이스에 대해 질의가 이루어지는 시맨틱 웹 환경에서는 매우 유용한 방법이다. 본 논문에서 제안하고 있는 기법은 프로그램 분석 없이 캐시를 진행시키는 것에 비해 캐시 대상 선정의 정확성이 높아지게 되어 궁극적으로 RDF 처리 프로그램들의 성능을 향상시킬 수 있다.

본 논문의 구성은 다음과 같다. Ⅱ장에서는 관련 연구에 대해 설명하고, Ⅲ장에서는 Jey언어의 구조와 특징에 관해 소개한 후, Jey언어에서의 RDF 표준 질의어인 SPARQL[8]의 적용 원리와 제안하는 분석 방법에 대해 설명한다. Ⅳ장에서는 성능 분석 결과를 소개하고, 마지막으로 Ⅴ장에서는 결론을 맺는다.

Ⅱ. 관련 연구

1. 질의 결과에 대한 캐시

앞서 언급한 바와 같이 RDF 질의의 경우 일반적으로 속도가 매우 느리고, 동일한 질의가 반복되어 실행되는 경우 서버 및 네트워크 성능의 저하를 초래하게 된다. 뿐만 아니라 RDF W3C 질의 표준인 SPARQL문법 중 'having', 'group by' 와 같은 비용이 큰 연산의 경우 프로그램의 전반적인 성능이 저하될 수 있다[9].

이를 해결하기 위한 것으로, 질의의 결과가 후에 재사용될 가능성이 있는 경우 캐시에 저장을 해두고, 후에 캐시에 저장된 결과를 이용하여 성능을 향상시키는 방법이 있다. 서로 다른 질의의 결과간의 포함관계를 질의문들의 구문에 의거하여 판별하고, 이에 따라 주어진 질의의 결과를 캐시에 보관할지 결정하는 절차가 필요하다. 이론적으로는 관계형 데이터 시스템을 위해 'materialized view'라는 이름으로 연구되어 왔으며[10~12], 그 성능 향상 정도가 입증되어 2000년 초반부터 Oracle 등 상용 시스템에도 도입이 되어 사용되고 있다[13].

이 방법은 프로그램에서 사용될 모든 질의와 수행 순서를 사전에 파악한 후 그 결과들 간의 포함관계를 잘 알고 있을 때 가장 유용하다. 역으로 이 방법의 한계는 앞으로 수행될 질의들에 대한 정보가 없거나 불분명한 상태에서 현재의 질의 결과를 캐싱할지 판단해야한다는 점이다. 본 논문에서는 응용 프로그램 전체를 미리 분석하여 수행 순서 및 포함관계를 판단해 놓는 기법을 사용하여 이와 같은 한계를 극복한다.

또한, RDF에 대한 질의를 SQL 문으로 변환하여 관계형 데이터 시스템에서 검색하는 경우 중간 결과를 잠시 동안 제한적으로 보관하게 되는데 이 개념도 캐시와 유사하다고 볼 수가 있다[14~15]. 그러나 이것은 일반적인 캐시 개념과 달리, 보관된 중간 테이블들은 후에 나올 질의를 위한 것이 아니라 해당 질의문 처리를 위해서만 제한적으로 사용되게 된다.

2. 기존의 RDF 질의 최적화 기법

캐시 이외에도 RDF 질의의 최적화에 대한 다른 연구들이 있어왔다. 최적화 방향이 다르므로, 이들 방법들은 본 논문에서 다루고 있는 캐시 방법과 상호보완적으로 적용될 수 있는 기술들이라고 할 수 있다. 또한 이와 같은 방법들은 본 연구에서 제안하는 RDF 질의의 캐시를 위한 질의 포함관계를 확장시키는 데에 단서 역할을 할 것으로 본다.

O. Hartig[16]등은 질의에 대상이 되는 RDF 데이터 집합을 사전에 줄여놓기 위해 데이터의 연결고리를 통해 관련 데이터를 비동기적으로 추출해두는 알고리즘을 제안하고 있다. 기반 데이터 집합을 줄이거나 미리 확보한다는 점은 캐시와 비슷한 개념이나, 별도의 추출 없이, 프로그램을 미리 분석하여 이미 질의 결과로 전달된 데이터를 사용한다는 점에서 본 논문에서 사용된 캐시 방법이 보다 효율적이라 할 수 있다.

A Bernstein[17] 등은 RDF 질의를 수행하기 위한 중간 결과들의 크기를 줄이는 시도를 하였다. 동일한 질의 내에서, 질의를 구성하는 서브연산의 순서를 선택율(selectivity)을 기반으로 조정한다. 그러나 이 방법은 현재의 선택율을 과거의 데이터에서 추론해 내거나 실시간에 알아내기 쉽지가 않다는 한계를 기본적으로 가지고 있다.

J. Zemánek[18] 등은 SPARQL의 문법을 확장하여 프로그래머가 특정형태의 질의문에 대해 미리 정보를 주어 이에 따라 처리함으로써 통신 부담을 줄이는 방법을 제안하고 있다. 이 경우 사용자의 개입을 필요로 한다는 점과 표준 문법을 확장해야한다는 단점이 존재한다.

3. 기존의 프로그램 분석과 데이터베이스 프로그램

프로그램의 정적 분석은 프로그램이 수행되기 전에 사전에 프로그램을 분석해내는 것으로 주로 컴파일러 제작에서 최적화 등을 위해 사용되는 기법이다. 이러한 정적 분석을 데이터베이스를 위한 응용 프로그램 분석에 적용된 연구는 초기에는 질의문과 성질이 비슷한 선언적 프로그래밍 언어 (declarative programming language)에서의 데이터 접근 분석에 대한 연구가 있었으며[19], 비교적 최근에는 SQL Injection 보안 취약점을 사전에 판별하기 위한 분석 기법들에 적용된다[12, 20~21].

B. Wiedermann[7]등은 프로그램 내에 데이터베이스에 저장될 데이터를 처리하는 코드와 일반 코드가 섞여 있고 잦은 함수 호출이 연루된 경우에 데이터베이스 관련 부분을 추출해 내는 분석 방법에 대해 연구하였다. Java와 HQL (Hibernate Query Language)[22]를 기반으로 분석의 정교성에 초점을 두고 있으며, 본 논문에서와 같이 캐시 등의 특정 최적화 방법을 위한 분석이 아닌 일반적인 분석이므로 추후 본 논문의 방법에 결합시킬 계획이다.

4. 기타

시맨틱 웹 처리를 위한 전용 언어에 대한 연구는 Workshop on Scripting for the Semantic Web 학회[5] 등을 중심으로 최근 몇 년간 활발히 이루어지고 있다. 전반적으로는 생산성 향상에 초점을 맞추고 진행되고 있으며 프로그램 분석을 통한 성능향상에 대한 내용은 아직 발표되지 않고 있다.

Ⅲ. 본 론

1. Jey : Groovy 기반의 RDF 처리 프로그래밍 언어

가. Jey 프로그래밍의 특징과 구조

Jey 언어는 스크립트 언어인 Groovy[23~24]에 RDF 데이터 모델의 조작을 문법적으로 지원하는 것을 도입한 것을 특징으로 하는 언어이다. RDF 온톨로지의 조작과 구축을 빠르고 단순화 할 수 있으며, 소스코드의 양을 줄이고, 작성하는 데이터 모델의 구조를 쉽게 파악할 수 있는 다양한 문법을 제공하고 있다. 기반으로 하는 언어인 Groovy는 Java와 호환이 되는 스크립트 언어로써 GRails[25]와 같은 웹 프로그래밍용 프레임워크도 제공되고 있어서 RDF 처리를 위한 Jey의 기반 언어로 선택되었다. 표 1과 표 2는 data model을 생성하고 검색하는 동일한 프로그램으로, 각각 Java와 Jena API[3]를 이용하여 개발하던 기존 방식으로 작성된 소스코드와 Jey 언어를 이용하여 작성된 소스코드를 보여주고 있으며, [6] 표에서 볼 수 있듯이 Jey를 이용한 작성 방식이 훨씬 간결하고 직관적임을 알 수 있다. 그 밖의 Jey의 경로표현식에 관한 보다 자세한 설명은 [6]에 있으며, SPARQL 지원을 위한 내용은 본 장의 다음 절과 다음 장에서 걸쳐 최근에 개선된 사항들을 포함하여 소개되고 있다.

표 1. Java와 Jena API로 작성된 소스 코드의 예
Table 1. An example of code using Java with Jena API.

```
String personURI = "http://somewhere/NakminChoi";
String schemaURI = "http://www.w3.org/2001/vcard-rdf/3.0";

Model data = FileManager.get().loadModel(personURI);
Model schema = FileManager.get().loadModel(schemaURI);
Model model = ModelFactory.createRDFSModel(schema, data);

ResIterator iter = model.listResourcesWithProperty(VCARD.FN);

while (iter.hasNext()) {
    System.out.println(iter.nextResource()
                       .getRequiredProperty(VCARD.FN)
                       .getString());
}  ...
```

표 2. Jey로 작성된 소스 코드의 예
Table 2. An example of Jey code.

```
def personURI = "http://somewhere/NakminChoi";
def schemaURI = "http://www.w3.org/2001/vcard-rdf/3.0";

def model  = new RDFmodel(personURI)
def schema = new RDFmodel(schemaURI)

def uri = schema.create( <personURI, "VCARD.FN"> )
model.schema(schema)

def search = model.objects().uri.findAll{println it}
```

Jey 언어는 다음과 같은 구조적 특징을 갖는다. Jey 소스코드를 작성한 후 컴파일을 하면 Jey 번역기에 의해 Groovy 형식의 코드로 재작성이 이루어지며, 해당 Groovy 코드는 Groovy 컴파일러에 의해 Java 바이트 코드로 변환되어 Java 가상머신 위에서 실행된다. 이러한 구조적 특징에 의해 Jey 코드 작성 시 Groovy의 문법 및 코드를 그대로 사용 할 수도 있으며, Java 클래스를 호출하여 사용할 수 있다는 장점이 있다.

나. Jey 프로그래밍 방법

RDF 데이터 관리의 주요 연산은 데이터 모델 및 데이터의 생성, 데이터 모델간의 관계 조작, 그리고 데이터에 대한 질의로 나누어 볼 수 있다. Jey 언어에서는 이러한 연산들이 기존의 API를 이용한 방식(e.g., 자바와 Jena API를 이용하는 방식)과 유사하게 메소드 호출을 통해 이루어진다. 하지만 이러한 메소드들의 일부 파라미터는 RDF의 트리플 집합을 직관적으로 표현하기 위해 특별한 형태로 표현되어 진다. 표 3은 RDF 모델과 트리플을 생성하는 코드를 보여준다. 키워드 *def* 는 Groovy에서 사용되는 것처럼 변수의 선언 및 정의를 의미한다.

표 3의 4번째 줄의 *RDFModel* 클래스는 기존의 전통적인 RDF API들에서 흔히 'model'이라 불리는 RDF graph를 재 표현한 것이다. 5번째 줄은 *RDFModel* 클래스의 메소드 호출을 통해 리소스를 만들고 model로 추가하는 코드를 보여주고 있다. 여기서 사용된 *create* 메소드의 파라미터는 "*<name_space, local_name>*"의 형태로 구성된다. 다음 6번째 줄의 코드의 의미는 *http://example.org/resource/#person1* URI의 리소스를 생성하는 것으로 5번째 줄과 동일하나, *create* 메소드의 세 번째 파라미터는 문자열 리터럴 값인 "Hong Gil-Dong"을 생성한 후, p_name이 가리키는 속성을 이용하여 리소스와 리터럴을 연결하는 것을 의미한다.

표 3. 간단한 Jey 코드의 예
Table 3. A example of the simple Jey code.

```
1: def ns = "http://example.org/"
2: def p_ns = ns + "property/"
3: def r_ns = ns + "resource/"

  /* create a model */
4: def model  = new RDFModel()
5: def p_name = model.create(<p_ns, "#name">)

  /* create resources */
6: def r_person1 =
          model.create(<r_ns, "#person1",
                 p_name:["Hong Gil-Dong"]>)
```

2. SPARQL 질의에 대한 분석

시맨틱 웹 환경에서는 원격지 서버의 성능과 네트워크 대역폭에 의해 질의 속도가 느려질 가능성이 매우 크므로 이전 질의 결과의 캐싱이 성능향상에 큰 도움을 줄 수 있다. 즉, 앞서 얻어진 질의의 결과가 후에 재사용될 가능성이 있는 경우 캐시에 저장을 해두면, 원격지로 질의를 전송하는 대신 캐시에 저장된 결과를 이용하여 성능을 향상시킬 수 있다. 또한, 원격지 서버 및 회선에 장애가 발생할 경우, 캐시를 이용하여 결과를 도출해 낼 수 있다는 장점이 있다. 그러나 Jena API에서 지원하는 ARQ[26]를 비롯한 이제까지 개발된 다양한 SPARQL 처리 방식에서는 캐시를 이용한 성능향상 기법에 대한 연구가 전무한 실정이다.

본 논문에서는 이러한 캐싱을 효과적으로 할 수 있도록, 컴파일 단계에서 소스코드의 문법을 분석하여 질의 결과를 추론한 후, 코드의 후반부에서 재사용이 발생하는지 여부를 판단할 수 있도록 하는 프로그램 분석 기법을 제안한다. 이를 위해 정적 분석을 통하여 어플리케이션에서 사용된 질의문들 사이의 유사성을 추론한 후, 재사용될 가능성이 있는 질의의 경우 결과를 캐시에 저장하여 성능을 높일 수 있도록 한다.

따라서 제안하는 기법은 다음의 세 가지 단계로 나뉜다.

1. 상관관계에 대한 정의 : 두 질의의 결과 간의 상관관계를 판별하는 방법을 정의한다. 특히, 질의 수행 전에 질의문 만으로 판별할 수 있도록 한다.

2. 프로그램 분석 단계 : 프로그램의 수행 전에 정적으로 프로그램을 분석하여 질의간의 상관관계 데이터를 추출한다.

3. 캐싱 단계 : 추출된 상관관계를 토대로, 원격으로 질의를 전송하는 대신 캐시에 있는 데이터에서 질의 결과를 추출한다.

간단한 예로, 표 4에서는, 10번 줄의 질의문과 25번 줄의 select 절과 where 절의 구성으로부터 10번 질의문이 15번 질의문을 포함하는 관계임을 알 수 있다. 따

표 4. 질의 최적화의 예
Table 4. An example of query optimization.

```
 ...
10: def query_string1 =
      "SELECT ?x ?name ?email" +
      "WHERE { ?x  <" + p_name + "> ?name ." +
             "?x  <" + p_email + "> ?email .}"
11: def qResult = model.querying(query_string1)
 ...
25: def query_string2 =
      "SELECT ?x ?name" +
      "WHERE { ?x  <" + p_name + "> ?name .}"
26: def qResult = model.querying(query_string2)
```

라서 11번 줄의 질의 결과를 캐시로 저장한 후, 26번 줄의 질의는 실제 데이터 대신 캐시에 대해 수행하도록 함으로써, 질의 성능을 향상시킬 수 있다. 비록 예제에서는 매우 간단한 예를 보여주고 있지만 이러한 방식은 데이터의 양이 크고, 질의문 간의 포함관계가 클수록 성능 향상의 폭이 커질 수 있다.

이어지는 절에서는 제안하는 기법의 각 단계에 대해 기술한다.

가. 질의간의 상관관계

프로그램 분석 단계에서는 질의 결과의 캐싱을 위하여 프로그램 코드 전반에 걸친 질의 문자열들의 상관관계를 분석한 후, 질의들 간 포함관계가 존재할 경우 수행한 질의 결과를 캐시에 저장한다. 따라서 정적으로 질의문의 텍스트로부터 결과의 상관관계를 알아내는 것이 필요하다.

이를 위해 먼저, 가능한 모든 RDF 트리플의 전체 집합을 T, 가능한 질의문 전체의 집합을 Q라고 하면, 함수 res: 2^T X $Q \rightarrow 2^T$는 특정 모델에 질의를 적용한 결과를 의미한다. 이는, 임의의 $m \in 2^T$, $q \in Q$에 대해, res(m, q)는 모델 m에 질의문 q를 적용하여 얻을 수 있는 결과인 트리플을 의미한다.

이를 토대로 두 개의 질의 간에는 다음과 같은 상관관계를 정의할 수 있다.

정의 1. [Subsumption]: q_1, q_2 $\in Q$ 에 대해 q_1이 q_2를 포함(subsume) 한다고 하는 것은 $res(q_2) \subseteq res(q_1)$ 을 만족함을 의미한다.

이러한 포함관계를 질의의 수행 전에 질의문만으로 판별하기 위해서는 프로그램을 정적분석 해야 한다. 우선 Jey 코드에서 제어흐름그래프 (control flow graph)와 기본 블록 Ibasic block) 들을 추출하고, 이 그래프를 고정 포인트 (fixed point) 방식으로 순회하면서 각 기본 블록마다 정보를 수집해가게 된다[27].

기본 블록은 전통적인 최대기본블록(maximal basic block)과 비슷하나, 한 기본 블록에 한 질의 구문 (표 4에서의 model.querying(...)) 만이 존재하도록 분할한다. 또한, 각 질의 구문에 바인딩 되는 질의 문자열에 대한 분석은 기존의 다양한 분석기법들을 사용하여 이미 이루어졌다고 가정한다. 표 4의 예에서는 11, 26 번 문장이 각각 질의를 수행하므로 최소한 두 개의 기본 블록이 존재하는 것을 볼 수 있다.

필터링과 추출을 수행하는 일반적인 질의의 포함 관계를 찾는 것은 간단한 알고리즘으로는 어려우며, 복잡한 경로 표현식을 포함하는 경우 아예 선형적으로 찾을 수 있는 알고리즘이 없다고 알려져 있다. 따라서 본 논문에서는 간단한 포함관계에 대한 판별을 수행하여 제안하는 방법의 적용 가능성을 보였다. 즉, 다음과 같은 규칙을 적용하여 판별하고 있다.

규칙 1. select 절을 추출되는 리소스명의 집합으로 보고, where 절을 트리플 패턴의 집합으로 볼 때, select 절간의 포함 관계와 where 절간의 포함관계를 동시에 만족시키는 경우 두 개의 질의가 포함관계에 놓여있다.

나. 프로그램 분석

먼저 각 기본 블록 X 마다 USE와 DEF 집합을 수집한다. DEF는 기본 블록 X에서 정의되는 질의문들을 모은 집합이다. USE는 이 프로그램에서 정의된 모든 질의 중 X에서 정의된 질의문을 포함(subsume) 하는 질의문들의 집합이다. 즉, USE(X)의 한 원소를 q'이라고 한다면, 이 q' 이 포함(subsume)하게 되는 어떤 질의문 q가 X 내에서 정의되고 있다는 것을 의미한다. 이 경우 q는 q'의 결과를 재사용하여 얻어낼 수 있다.

$$\begin{aligned}
&\textbf{for } \text{'query}(q)\text{' } \textbf{in X do} \\
&\quad \text{DEF(X)} = \text{DEF(X)} \cup \{q\} \\
&\quad \text{USE(X)} = \text{USE(X)} \cup \{q' \mid q' \in Q \\
&\qquad\qquad \textit{which subsumes } q \textit{ and } q' \neq q\}
\end{aligned} \tag{1}$$

이 식에서 USE는 결국 주어진 프로그램의 모든 질의간의 포함관계를 조사하는 작업을 수반하게 된다.

위 식에서 구한 USE/DEF 집합의 정보를 가지고 제

어 흐름 그래프를 순회하여 블록 간의 정보를 합하면 다음 식 (2)와 같이 IN/OUT 집합을 구할 수 있다[27].

```
initialize IN(X) to NULL for all basic blocks X
change = 1
while(change) do
    change = 0;
    for each basic block, X, do
        old_IN = IN(X)
        OUT(X) = ⊔ {IN(Y)| Y ∈ successor(X)) }      (2)
        IN(X) = USE(X) + OUT(X) - DEF(X)
        if (old_IN != IN(X)) then
            change = 1
        endif
    endfor
endwhile
```

(2)번 식에서 IN(X)는 블록 진입 시 계산되어지며, 진입되는 블록 X 내에서 사용되는 질의와 뒤따라오는 다음 블록들에서 사용될 질의들의 합집합을 의미한다. 또한, OUT(X)는 뒤따라오는 다음 블록 (successor)들에서 질의 사용 정보(IN(Y))의 집합을 의미한다.

예를 들어 표 5의 Jey 코드의 기본 블록과 제어흐름 그래프는 그림 1과 같이 나타낼 수 있다. 각 기본 블록에 대해 USE/DEF, IN/OUT을 추출한 정보는 표 6과 같으며, #0~#3은 각 기본 블록을 나타낸다.

기본 블록 X에서 정의되는 질의문 중에서 OUT(X)가 존재하면 그 질의문은 캐시 되어야한다. 즉, 기본 블록 #1에서 정의되는 res1은 res3에서 사용될 수 있으므로, 캐시 되었을 때 성능 향상을 볼 수 있다. 캐시에 보관될 질의문은 해당 블록 X에서 정의되는 것 중 차후에 사용될 가능성이 있는 것이다. 따라서 다음 정의 2와 같은 기준으로 선정 된다.

표 5. SPARQL 코드의 예
Table 5. An example of SPARQL code.

```
def p_person = "http://example.org/PInfo"
def p_company = "http://example.org/CInfo"
def model = new RDFModel()
def selectSQL = null

switch (QUERY_TYPE) {
 case:GET_INFO_ALL
  selectSQL =
   "SELECT ?name ?email ?office ?addr " + "where { "+
    "?person <"+ p_person +"#fullName> ?name. "+
    "?person <"+ p_person +"#mailbox> ?email. "+
    "?company <"+ p_company +"#name> ?office. "+
    "?company <"+ p_company +"#addr> ?addr. }"
  def res1 = model.querying(selectSQL);
 break;

 case:GET_INFO_PERSON
  selectSQL =
   "SELECT ?name ?email" + "where { "+
    "?person <"+ p_person +"#fullName> ?name. "+
    "?person <"+ p_person +"#mailbox> ?email. }"
   def res2 = model.querying(selectSQL);
   break;
}

selectSQL =
   "SELECT ?name ?office " + "where { "+
   "?person <"+ p_person +"#fullName> ?name. "+
   "?company <"+ p_company +"#name> ?office. " }"
def res3 = model.querying(selectSQL);
```

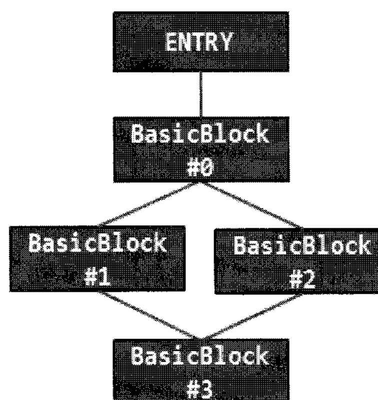


그림 1. 표 5의 제어흐름 그래프
Fig. 1. Control Flow Graph of the table 5.

표 6. 추출된 정보
Table 6. Extracted information.

	USE	DEF	IN	OUT
#0	∅	∅	{res1}	{res1}
#1	∅	{res1}	∅	{res1}
#2	∅	{res2}	{res1}	{res1}
#3	{res1}	{res3}	{res1}	∅

정의 2. 기본 블록 X에서 캐시에 보관될 질의문 CASH(**X**)는 다음과 같이 나타내어진다.

CASH(**X**) = $\{ q \mid q \in$ OUT(**X**) $\cap$ DEF(**X**) $\}$

그런데, 기본 블록마다 반드시 하나 이하의 질의문 정의를 내포하도록 블록을 분할하였으므로 이것은 다음과 같이 바꾸어 표현할 수 있다.

정의 2′. 기본 블록 X에 대해, OUT(**X**) $\cap$ DEF(**X**) $\neq \emptyset$ 라면, X에서 정의된 질의문은 캐시 된다.

다. 캐싱 단계

캐시의 자료구조는 hashtable을 이용하여 구현하였다. 이때 저장될 key 값은 DEF 집합에서 정의된 변수와 변수의 술어로 하였으며, 둘 사이는 세미콜론(;)으로 구분되어진다. 또한 각 key 별로 질의 결과인 Model을 저장하여 검색의 효율성을 높였다. 표 5에서 사용된 첫 번째 질의문은 표 7과 같이 나타낼 수 있다.

질의를 하기 위한 알고리즘은 표 8과 같이 구성된다. 첫 번째 줄은 질의문을 검사한 후, SELECT,

표 7. hashtable의 예
Table 7. An example of hashtable.

Key	Value
name; http://example.org/PInfo#fullName	*Model_1*
email; http://example.org/PInfo#mailbox	*Model_2*
office; http://example.org/PInfo#name	*Model_3*
addr; http://example.org/PInfo#addr	*Model_4*

표 8. 질의 최적화에 대한 기본 알고리즘
Table 8. Basic Algorithm of query optimization.

```
 1: switch(queryType) {
 2: case SELECT:
 3: case DESCRIBE:
 4: case CONSTRUCT:
 5:   if(isCacheUsable == true)
 6:     queryToCache();
 7:   else {
 8:     query();
 9:     if (isQueryUsingNextTime == true)
10:       saveToCache();
11:   }
12:   break;
13: case ASK:
14:   executeASK();
15:   break;
16: }
```

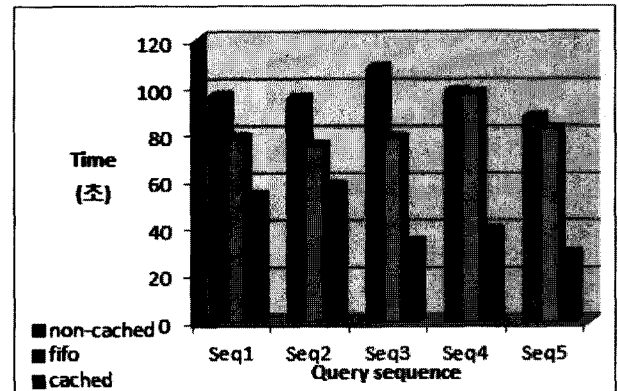
DESCRIBE 또는 CONSTRUCT 형식일 경우 현재 수행될 질의가 캐시를 이용할 수 있는지를 확인하여(5번째 줄), 가능한 경우 캐시로 질의 하도록 한다. 그렇지 않은 경우에는 일반적인 질의를 수행한 후, 결과 값이 프로그램의 후반부에서 다시 사용될 경우에만(DEF 집합을 이용) 캐시에 저장하도록 한다. ASK 질의문은 true 또는 false 값을 반환하므로 따로 캐시에 저장하지 않는다.

IV. 성능 평가

본 실험은 64비트 Windows 환경의 4G 메인 메모리와 인텔 쿼드코어 2.4GHz CPU를 탑재한 컴퓨터에서 수행하였으며, 64bit 자바 가상 머신 위에서 Jey로 구현되었다. 온톨로지 데이터를 제공하는 서버는 Sun OS 5.9가 설치된 Sun Fire v880 환경에서 구축하였으며, 네트워크는 동일한 subnet에 100Mbps로 연결되어있다. 실험에 사용한 데이터는 단백질 자원 정보를 가장 포괄적으로 다루고 있는 Uniprot knowledge base를 대상으로 하였고, 트리플 집합의 수는 약 30만 개로 제한하였다.

그림 2는 포함관계가 50%정도인 임의의 질의문 10개를 생성 후, 프로그램 내에서 질의의 순서를 바꾸어가며, 제안된 기법을 적용한 경우와 적용하지 않은 수행 결과를 비교하고 있다. 여기서는 질의 순서가 다른 프로그램을 Seq1~Seq10이라 명명하고 있다.

그래프의 왼쪽 막대는 캐시가 적용되지 않은 것이며, 가장 오른쪽 막대는 프로그램 분석을 통해 캐시하는 기법이 적용된 것이다. 가운데 막대는 프로그램 분석 정보 없이 FIFO 방식으로 캐시하는 것에 대한 결과를 보여주고 있다. 캐시를 사용한 경우는 캐시를 사용하지 않은 경우보다 대부분 성능이 우수한 것으로 나타났다. 예를 들어 Seq9의 경우, 제안된 기법을 적용하여 캐시를 사용한 경우 18sec가 소요되어 캐시를 사용하지 않은 경우 99sec 소요한 것 보다 5.5배 더 빨랐다. 반면 Seq7의 경우 먼저 수행된 질의가 뒤에 수행된 질의를 포함하는 부분이 전혀 없도록 배열되었기 때문에 캐시를 위한 오버헤드에 의해 성능이 오히려 떨어지는 결과를 보여주었다. 또한 질의 처리를 위해 소요되는 122초 중 2~3초가량이 캐시를 운용하기 위해 걸린 시간임을 알 수 있다.



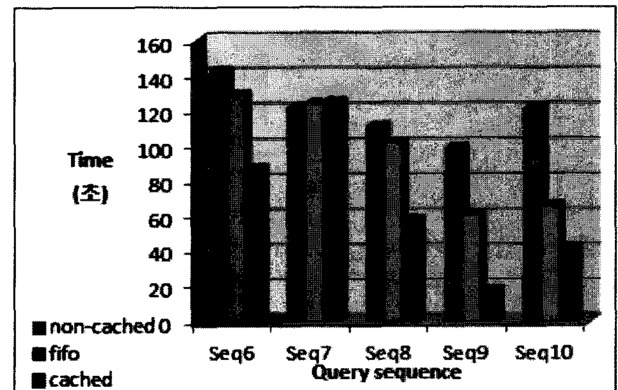


그림 2. 캐시 사용 여부에 따른 성능 비교
Fig. 2. Performance comparison between cached with non-cached.

표 9. 성능 비교 표
Table 9. Performance comparison table.

Query sequence	Seq1	Seq2	Seq3	Seq4	Seq5
cached	54	58	34	39	29
FIFO	78	75	78	97	82
non-cached	96	95	108	98	87
Query sequence	Seq6	Seq7	Seq8	Seq9	Seq10
cached	87	125	58	18	42
FIFO	129	124	102	61	66
non-cached	142	122	111	99	121

FIFO 형태로 캐시를 운영한 경우와 비교했을 때에도 제안하는 방식으로 캐시를 운영한 경우가 전반적으로 더 우수하였다.

제안하는 기법에서 처리 가능한 질의의 배열을 보다 더 포함하고 있는 Seq3, Seq9, Seq10 등에서는 성능향상이 크게 나타났으며 그렇지 않은 경우 (Seq7)에서는 성능이 비슷하게 나타났다. 따라서 본 논문에서 제안하는 기법에 따른 성능 향상을 극대화 시킬 수 있는 방법은 보다 폭넓은 질의간 포함관계를 밝혀내는 것을 수반하게 된다. 이러한 연구는 관련 연구에서 언급된 여러 RDF 최적화 기법들을 단서로 하여 발전 가능할 것으로 보이며 본 연구의 향후 연구 주제이기도 하다.

결과적으로 정리하면 이 실험 결과에서 알 수 있듯이 질의들 간의 포함관계가 많이 존재할수록 캐시에 저장할 수 있는 양이 많아지며, 따라서 질의 성능이 향상되는 것을 알 수 있다. 또한 프로그램 분석을 통해 캐시를 정확히 하는 것도 그렇지 않은 FIFO 방식에 비해 성능향상에 적지 않게 기여하게 된다는 것을 알 수 있다.

V. 결 론

본 논문에서는 기존 RDF 데이터 처리를 위한 프로그래밍언어에서의 질의 방법을 확장하고 질의 성능을 높이기 위한 방법으로 프로그램 정적 분석을 통해 재사용 가능한 질의 결과를 캐시에 저장하여 사용하는 방법을 제안한다. 먼저 Jey 언어[6]와 SPARQL[8]을 사용하는 프로그램에 대해 USE/DEF 집합과 IN/OUT 집합의 분석을 수행하여 질의문들 간의 상관관계를 파악하였다. 만일 응용 프로그램의 앞부분에서 쓰인 질의 결과의 일부 혹은 전체가 뒤에서 작성된 질의에 포함될 경우, 앞서 수행되는 질의 결과를 캐시에 저장하도록 한다. 후에 수행되는 질의는 실제 웹상의 자원에 접근하는 대신 캐시로부터 결과를 받아올 수 있게 되므로 성능을 향상시킬 수 있다.

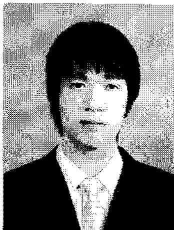
현재는 앞서 언급되었듯이, 보다 정교하게 폭넓은 포함 관계를 정의할 수 있는 규칙들을 규명하기 위해 연구하고 있다. 또한 기존의 다양한 정적 분석 기법을 활용하여 인자가 있는 경우와 반복문이 있는 경우에도 효율적으로 수행될 수 있는 정적 분석 구조에 대해서도 연구할 계획이다.

참 고 문 헌

[1] H. Chen, T. Finin and A. Joshi, "Semantic Web in a Pervasive Context-Aware Architecture", Artificial Intelligence in Mobile System, pp33-40, Seattle, USA, Oct. 2003.
[2] Gomez-Perez, A. and Corcho, O., "Ontology languages for the Semantic Web", IEEE Intelligent Systems, Vol. 17, No. 1, pp.54-60, Jan./Feb. 2002
[3] Jena, http://jena.sourceforge.net/
[4] E. Oren et. al, "ActiveRDF: ObjectOriented Semantic Web Programming", WWW, pp.817-824, Alberta, Canada, May. 2007.
[5] Semantic Scripting, http://www.semanticscripting.org/
[6] Ik-Hyun Jhin, Nak-Min Choi, Eun-Sun Cho, "Integration of RDF Processing into a Programming Language" NCM, pp.196-199, Seoul, Korea, Aug. 2009.
[7] Ben Wiedermann, Ali Ibrahim, William R. Cook, "Interprocedural query extraction for transparent persistence", OOPSLA, pp.19-36, Nashville, USA, Oct. 2008.
[8] SPARQL Query Language for RDF, W3C, http://www.w3.org/TR/rdf-sparql-query/
[9] Edward Hung, Yu Deng, V.S. Subrahmanian, "RDF Aggregate Queries and Views", ICDE '05, pp.717-728, Washington, DC, USA, Apr. 2005.
[10] Yao-Wen Huang, Fang Yu, Christian Hang, Chung-Hung Tsai, Der-Tsai Lee, Sy-Yen Kuo, "Securing web application code by static analysis and runtime protection", WWW, NY, USA, May. 2004.
[11] Divesh Srivastava, Shaul Dar, H. V. Jagadish, Alon Y. Levy, "Answering SQL Queries Using

Materialized Views", VLDB, Seoul, Korea, Sep. 2006.
[12] Alon Y. Halevy, Theory of answering queries using views, ACM SIGMOD Record, vol29, no.4, Dec. 2000.
[13] Lilian Hobbs, "Oracle9i Materialized Views: An Oracle White Paper", http://www.oracle.com/technology/products/oracle9i/pdf/o9i_mv.pdf, May 2001.
[14] Eugene Inseok Chong, Souripriya Das, George Eadon, Jagannathan Srinivasan, "An Efficient SQL-based RDF Querying Scheme", VLDB, pp.1216-1227, Trondheim, Norway, Sep. 2005.
[15] Daniel J. Abadi, Adam Marcus, Samuel R. Madden, Kate Hollenbach "Scalable semantic web data management using vertical partitioning", VLDB, Vienna, Austria, Sep. 2007.
[16] Olaf hartig, Christian Bizer, Johann-Christoph Freytag, "Executing SPARQL Queries over the Web of Linked Data", ISWC, pp.293-309, VA, USA, Oct. 2009.
[17] Abraham B., Christoph K., Marjus S., "OptARQ: A SPARQL Optimization Approach based on Triple Pattern Selectivity Estimation", Technical Report No, ifi-2007.03, Mar. 2007.
[18] Jan Zemánek, Simon Schenk, Vojtech Svatek, "Optimizing SPARQL Queries over Disparate RDF Data Sources through Distributed Semi-Joins", ISWC, Karlsruhe, Germany, Oct. 2008.
[19] Gupta A, Harinarayan V, Quass D, "Aggregate-query processing in data warehousing environment", VLDB, pp.358-369, Zurich, Switzerland, Sep. 1995.
[20] MS Lam et.al, Context-sensitive program analysis as database queries, PODS05
[21] Stephen Thomas, Laurie Williams, Tao Xie, "On automated prepared statement generation to remove SQL injection vulnerabilities", Information and Software Technology, volume 51, number 3, pp.589-598, Mar. 2009.
[22] Hibernate Query Language, http://docs.jboss.org/hibernate/core/3.3/reference/en/html/queryhql.html/
[23] Groovy, http://groovy.codehaus.org/GPath/
[24] K. Barclay and J. Savage, "Groovy Programming and introduction for Java developers", Morgan Kaufmann Pub., 2007.
[25] Grails - The search is over, http://www.grails.org/
[26] ARQ, http://jena.sourceforge.net/ARQ/
[27] Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman, "Compilers: Principles, Techniques, and Tools, 2/E", Addison-Wesley, pp.597-617, Dec. 2007.

저 자 소 개



최 낙 민(학생회원)
2009년 충남대학교 컴퓨터공학과 학사 졸업.
2009년~현재 충남대학교 컴퓨터공학과 석사과정.
<주관심분야 : 컴파일러 디자인, 시맨틱 웹, 유비쿼터스 컴퓨팅>

조 은 선(정회원)-교신저자
1991년 서울대학교 계산통계학과 학사 졸업.
1993년 서울대학교 계산통계학과 전산과학전공 석사 졸업.
1998년 서울대학교 계산통계학과 전산과학전공 박사 졸업.
1999년~2000년 한국과학기술원 연구원.
2001년 미국 Carnegie Mellon University SEEK (Software Engineering Evangelist of Korea) 이수 (정통부지원).
2002년~2006년 충북대학교 전기전자컴퓨터공학부 교수.
2006년~현재 충남대학교 컴퓨터공학과 교수.
<주관심분야 : 프로그래밍 언어, 컴파일러, 유비쿼터스 컴퓨팅, 시맨틱웹>