XML 질의의 수행성능 향상을 위한 트리 구조 XPath 질의의 축약 기법에 관한 연구

이 민 수[†] · 김 윤 미[††] · 송 수 경[†††]

요 약

일반적으로 XML 데이터는 트리 형태의 계층적인 구조를 가지고 있으며, XML 데이터의 저장 및 검색도 이러한 특성을 반영한다. 따라서 XML 데이터를 데이터베이스화 할 때에 XML 엘리먼트 간의 이러한 계층 관계를 반영하여 XML 데이터를 구조화하여 저장하고, 사용자의 검색을 지원하기 위해서는 질의에 명세 된 엘리먼트 구조 간의 계층 관계를 계산하여 처리하는 방법이 필요하다. 구조적 조인(structural joins) 연산은 이 문제의 한 해결책으로서 노드 번호 매기기 방식(node numbering scheme)에 기반한 XML 데이터베이스에 대하여 효율적인 계층 관계 연산 기법을 제시하고 있다. 하지만 계층 관계가 복잡하게 중첩되어 있는 트리 구조의 XML 질의를 처리하려면 여전히 다수의 구조적 조인을 수행해야 하기 때문에 질의 처리 비용이 많이 드는 또 다른 문제를 갖게 된다. 이에 본 논문에서는 선행 연구에서 제안된 트리 구조의 XML 질의 처리시에 필요한 다수의 중첩된 구조적 조인들의 수행비용을 효과적으로 줄이기 위한 사전 처리 방법으로서 동등 클래스 개념을 적용한 정규 표현식(regular expression)으로 된 경로 질의(path query)의 길이를 단축하는 경로식 단축 알고리즘을 소개하며 특히 분기 노드(branch node)가 포함된 경로식 단축 알고리즘을 제안한다. 제안한 알고리즘이 XML 경로식 질의 처리 시간을 평균적으로 1/3로 단축할 수 있음을 실험을 통해서 확인한다.

키워드 : XML, XPath, 트리 구조 질의, 질의 축약

A Tree-structured XPath Query Reduction Scheme for Enhancing XML Query Processing Performance

Lee, Minsoo[†] · Kim, Yun-mi[††] · Song, Soo-kyung[†††]

ABSTRACT

XML data generally consists of a hierarchical tree-structure which is reflected in mechanisms to store and retrieve XML data. Therefore, when storing XML data in the database, the hierarchical relationships among the XML elements are taken into consideration during the restructuring and storing of the XML data. Also, in order to support the search queries from the user, a mechanism is needed to compute the hierarchical relationship between the element structures specified by the query. The structural join operation is one solution to this problem, and is an efficient computation method for hierarchical relationships in an XML database based on the node numbering scheme. However, in order to process a tree structured XML query which contains a complex nested hierarchical relationship it still needs to carry out multiple structural joins and results in another problem of having a high query execution cost. Therefore, in this paper we provide a preprocessing mechanism for effectively reducing the cost of multiple nested structural joins by applying the concept of equivalence classes and suggest a query path reduction algorithm to shorten the path query which consists of a regular expression. The mechanism is especially devised to reduce path queries containing branch nodes. The experimental results show that the proposed algorithm can reduce the time required for processing the path queries to 1/3 of the original execution time.

Key Words : XML, XPath, Tree-structured Query, Query Reduction

1. 서 론

XML이 웹상에서 정보 교환의 표준 언어로써 효용성이 높아짐에 따라, XML 데이터에 대한 저장 및 검색 기술들은 점점 더 중요한 기술로 인식되고 있다. 일반적으로 XML 데이터는 트리 형태의 계층적인 구조를 가지고 있어서 XML 데이터를 데이터베이스화 할 때에는 XML 엘리먼트 간의 계층 관계를 반영하여 XML 데이터를 구조화하고 저장한다. 또한 사용자의 검색에 있어서도 질의의 엘리먼트 구조 간의 계층 관계를 계산하여 처리하는 방식이 필요하다.

※ 이 논문은 2단계 BK21 사업에 의하여 지원되었음.
† 종신회원 : 이화여자대학교 컴퓨터학과 교수
†† 정 회 원 : 이화여자대학교 컴퓨터학과 석사
††† 준 회 원 : 이화여자대학교 컴퓨터학과 석사과정
논문접수 : 2007년 5월 22일, 심사완료 : 2007년 8월 8일

XML 데이터를 트리 형태로 표시할 때는 노드 번호 매기기 방식(node numbering scheme)으로 XML 엘리먼트 간의 계층 관계를 표시할 수 있다. 즉, 트리를 순회할 때 XML 엘리먼트를 나타내는 각 노드는 <DocId, StartOffset, EndOffset, Level> 정보로 표시한다[1, 2]. DocId는 XML 문서가 하나 이상일 때 구분하는 문서 식별자이고, StartOffset과 EndOffset은 해당 XML 엘리먼트의 문서상에서의 시작 Offset과 끝 Offset이며, Level은 해당 엘리먼트의 루트노드로부터의 깊이를 나타낸다. 이러한 방법을 통해 엘리먼트 간 계층 관계는 조상 자손(ancestor-descendent) 관계와 부모 자식(parent-child) 관계로 구분할 수 있다.

위의 방법을 통해 사용자가 질의에서 요구하는 엘리먼트 간의 계층 관계를 해석하는 연산을 구조적 조인(Structural join)이라고 정의하는데, 구조적 조인은 일반적인 데이터베이스에서 지원하는 동등 조인(equality join)과는 달리, 프레디키트가 여러 개이고 비동등 조인(inequality join)이라는 특징을 가지기 때문에 데이터베이스에서 구현되어 있는 일반적인 조인 알고리즘들을 적용하기에는 비효율적이다. 최근에 다수의 연구 논문에서 XML의 구조적 조인을 효율적으로 처리하는 알고리즘[3] 및 인덱싱 기법[4, 5]을 제안한 바 있다.

그런데 제안된 구조적 조인 방법들이 XML 엘리먼트 간의 계층관계를 효율적으로 처리하는 문제에 대한 해결책들을 제시하고 있지만, XML 질의 처리기 입장에서는 계층 관계가 중첩되어 있는 트리 구조의 XML 질의를 처리하려면 여전히 다수의 구조적 조인을 수행하여야 하기 때문에 질의 처리 비용이 여전히 많이 드는 문제를 갖는다. 이러한 문제를 완화시키는 질의 최적화의 한 방법으로 중첩된 구조적 조인들에 대한 조인 순서화(join ordering) 기법[6]이 있다. 이것은 다수의 구조적 조인들의 순서를 질의 처리 비용이 최소가 되도록 정하는 방법이다. 그러나 조인 순서화 방법도 구조적 조인의 횟수가 많아지면 실제 질의 처리 비용을 줄이는 데에는 한계가 있게 된다.

이에 본 논문에서는 트리 구조의 XML 질의 처리 시에 필요한 다수의 중첩된 구조적 조인들의 수행비용을 효과적으로 줄이기 위해 구조적 조인 횟수를 줄이기 위한 XML 질의 축약 방법에 대해 연구하고자 한다. 이 방법은 사용자의 XML 질의에서 유도되는 노드 간의 계층 관계의 개수를 최소화 하는 알고리즘을 개발하여 질의 처리에서 필요한 구조적 조인의 횟수 자체를 줄이는 방법이다.

본 논문에서 제안하는 방법은 기존의 다항 복잡도를 갖는 질의 재작성 방식들에 비해 트리 구조의 질의 축약시에 동등 클래스 개념을 사용하면서 상향식(Bottom-Up)으로 알고리즘이 수행되어 초기에 고려대상에서 많은 탐색공간을 줄임으로써 축약시간의 성능이 우수하며 질의에 대한 분할과 병합시에 중복노드 제거와 같은 처리도 수행함으로써 한단계 더 축약된 질의 형태를 생성하도록 한다. 일반적인 XML 스키마들의 구조를 반영한 DTD나 XML Schema를 사용하지 않고 개별적인XML 문서의 구조를 반영한 XIP 트리를 사용하여 개별 문서에 대한 최적화가 더욱 효율적으로 진행되며 조건문을 포함한 트리 구조의 질의 축약을 지원할 수 있다. 실험을 통하여 경로식 질의 수행시간을 약 1/3로 줄일 수 있음을 확인하였다.

본 논문의 구성은 다음과 같다. 2장에서는 관련연구로 XPath에 대해서 그리고 노드 번호 매기기 방식에 대해 설명하고 XML 문서의 데이터 구조를 요약하는 기법을 제공하는 XML 문서 인덱스와 구조적 조인 기법에 대해 설명한다. 3장에서는 XPath 질의 축약 기법에 대하여 설명하고 알고리즘의 두 가지 접근법에 대해 소개한다. 그리고 4장에서는 본 논문에서 제안하고자 하는 트리 기반의 XPath 질의 축약 기법에 대해 설명하고 이 알고리즘의 두 가지 과정에 대해 언급한다. 5장에서는 트리 기반의 XML 질의 축약 알고리즘을 기초로 구현한 시스템에 대해 설명하였고, 이 시스템을 이용한 실험 및 성능에 대해 언급하였다. 마지막으로 6장에서는 결론 및 향후 연구 방향을 제시한다.

2. 관련 연구

2.1. XPath

XPath(XML Path Language)는 XML 문서의 노드 내에 포함된 정보를 검색하는 데 사용되는 질의 언어이다. XPath query는 식으로 구성된다. 이러한 식을 사용하여 XML 문서의 다양한 부분을 처리하고 문자열, 숫자 및 부울 값을 조작하고 문서 내에서 기준에 맞는 노드 집합을 찾는다.

XPath는 XML 문서를 여러 노드 형식으로 구성된 트리 구조로 인식한다. XPath 식은 XML 문서에 들어 있는 이러한 노드를 형식, 이름 및 값뿐만 아니라 문서에 있는 다른 노드와의 관계에 따라 식별하며 기본 개체 중 하나를 결과로 반환한다.

XPath 식은 컨텍스트 노드를 기준으로 노드나 노드 집합의 상대적인 위치 경로를 지정하는 패턴으로 구성된다. 컨텍스트 노드는 XML 문서에서 사용자의 현재 위치를 나타내는 섹션이며 XPath에서는 마침표(.)로 표시한다. XPath에서는 문서 루트를 기준으로 절대 위치를 지정하는 패턴도 사용할 수 있다. 문서 루트는 XPath에서 슬래시(/)로 표시한다. XPath는 위치 경로 외에 XML 문서에서 데이터를 검색하는 데 사용할 수 있는 다양한 함수를 제공한다[7].

2.2. 노드 번호 매기기 방식

관계형 데이터베이스에서 XML 데이터를 저장하는 방법 중에서 가장 널리 알려진 방법이 노드 번호 매기기 방법이다. 이는 XML 문서상의 각 노드(element 혹은 attribute)를 관계형 테이블의 투플(tuple)로 자연스럽게 매핑할 수 있기 때문일 것이다. 각 투플은 <DocId, StartOffset, EndOffset, Level>의 구조로 표현된다. DocId는 XML 문서가 하나 이상일 때 구분하는 문서식별자이고, StartOffset과 EndOffset은 해당 XML 엘리먼트의 문서상에서의 시작 오프셋과 끝 오프셋이며, Level은 해당 엘리먼트의 루트노드로부터의 깊이를 나타낸다. 여기에서 엘리먼트 간 계층 관계는 2가지로

구분할 수 있다. 첫째로 두 엘리먼트 A와 B가 조상 자손 관계에 있다는 것(XPath 표기: A//B)은 조건 "A.DocId = B.DocId AND A.StartOffset <B.StartOffset AND A.EndOffset > B.EndOffset"을 만족한다는 것을 의미하고, 둘째로 두 엘리먼트 A와 B가 부모 자식(parent child) 관계에 있다는 것(XPath 표기: A/B)은 위의 조상 자손 관계에 대한 조건에 "A.Level + 1 = B.Level"을 추가한 조건을 만족한다는 것이다.

노드 번호 매기기 방법은 입력 데이터에 대하여 DTD 혹은 XML 스키마 등의 정보가 없어도 일반적인 XML 데이터를 저장하고 검색할 수 있다는 장점을 가진다. 하지만, 이 방법은 몇 가지 단점이 있다. 첫째, XML 검색 시, 필수 요소라 할 수 있는 경로식(path expression)을 처리하는데 경로식의 길이만큼의 횟수의 조인이 필요하다. 이러한 연산은 메모리 부족을 야기 시켜 빈번한 입출력(I/O)을 발생시키기 때문에 성능 저하가 심각해질 수 있다. 또한, XML 데이터의 복원 측면에서도 마찬가지의 결과를 가져온다. 둘째, 갱신 연산시에 자식 노드의 갱신이 부모 노드의 갱신을 야기시킬 수 있기 때문에 효율적이지 못하다는 단점을 갖는다.

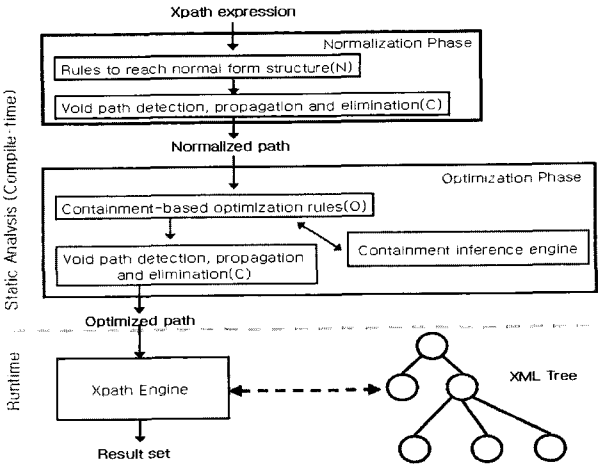
2.3 구조적 조인 기법

구조적 조인 알고리즘이란 주어진 후보 리스트에서 효과적으로 구조적 관계를 만족하는 쌍들을 찾아내는 알고리즘을 말한다. Chien[7]은 2002년에 확장된 선순위 번호 매기기 방식(Extended Preorder Numbering Scheme)과 B+ 트리를 이용하여 Al-Khalifa의 스택을 이용한 구조적 조인 알고리즘을 개선하였다. 영구적인 식별자가 지정되어 있고, 시작 위치에 따라 정렬된 엘리먼트 리스트들이 Chien의 알고리즘의 입력 값이 되며, 조상-후손 관계를 갖는 순서쌍들을 출력 값으로 반환한다. 이외에도 다양한 구조적 조인 알고리즘들이 제안되었다[8, 9].

2.4. XPath 최적화

XPath를 최적화하는 알고리즘으로 기존에 논리 기반(Logic-based)의 XPath 최적화 기법[10], 구조 기반(Schema-based)의 XPath 최적화 기법[11], 트리 패턴 질의 최소화 기법[12]들이 연구되어 왔다. 논리 기반의 XPath 질의 최적화 기법은 2단계를 거쳐 수행된다. 첫 번째 단계로는 2개의 규칙 집합을 이용하여 질의 경로식을 정규 경로(normal path)로 재작성 하는 정규화 단계고 두 번째 단계로는 앞의 단계에서 정규화된 정규 경로를 최적화 규칙 집합을 이용하여 변환하는 과정이다. 이 기법은 포함(containment) 관계를 이용하여 최적화 규칙들을 생성한다. 최적화 규칙은 중복 제거하는 규칙들의 집합과 빈 경로식(void path)을 제거하는 규칙들의 집합의 두 가지 집합으로 구성되어 있다. (그림 1)은 논리 기반의 XPath 변환 과정을 나타낸다[10].

두 번째 구조 기반의 질의 최적화 기법[11]은 Kwong et al. 에 의해 제안된 기법으로 DTD를 기반으로 하여 DTD에 존재하는 XPath 표현에 대해서도 논리적인 최적화가 가능하다는 장점을 가지고 있다. 이 기법은 XML문서의 경로들



(그림 1) 논리 기반의 질의 최적화 시스템의 구조

을 경로 동등 클래스(path equivalence classes)들로 나누는 것을 기반으로 하여 수행된다. 경로 동등 클래스는 같은 의미를 나타내는 경로들은 하나의 클래스에 속하도록 정의된다. 이는 본 논문에서 이용한 동등 클래스의 정의와 유사하나 본 논문에서는 XIP 트리를 기반으로 하여 동등 클래스를 작성하나 구조 기반의 XPath 축약은 DTD를 이용하여 동등 클래스를 형성시킨다는 점이 다르다. 또한 본 논문에서는 동등 클래스를 축약된 경로가 원래의 경로와 일치하는지를 평가하는데 사용하지만 이 기법은 동등 클래스를 질의를 최적화하는 규칙(rule)을 형성하는데 사용한다. 구조 기반의 최적화 기법은 앞의 과정에서 나누어진 경로들 중 최적화를 위해 몇몇의 경로들을 선택하기 위해 XTrie라고 불리는 구조를 생성한다[12]. XTrie는 DTD에 적합한 모든 경로들을 열거하는 구조를 가지고 있다. XTrie는 <S, S0, E, slabel, elabel, etype>의 6개의 투플로 구성되어 있다. S는 상태 집합을 나타내고, S0는 시작 상태, E는 간선의 집합을 뜻하며, slabel과 elabel은 각각 상태 레이블과 간선 레이블을 나타내고, etype은 E가 선택적인지, 필수적인지를 나타낸다. 이렇게 표시된 경로들은 정보를 통합하기 위한 마지막 과정으로 XTrie를 효과적으로 구조화 할 수 있도록 CIC-Graph(Class Implication and Contradiction graph)를 형성한다. 이렇게 형성된 그래프를 최적화 알고리즘을 통해 최적화해서 그래프를 재구성하므로써 경로를 최적화 한다.

트리 패턴 질의 최소화 기법[13]은 통합 규칙(integrity constraints)의 존재 여부에 따라 두 가지 방식의 질의 최소화 기법을 제안한다. 통합 규칙이 없을 때는 CIM 알고리즘을 이용하여 질의를 최소화 한다. CIM 알고리즘은 두 가지의 주요한 특성을 가지고 있다. 하나는 각 노드는 자식 노드들이 없다면 중복될 수 없다는 것이고 다른 하나는 중복된 노드를 제거하는 순서는 관계없다는 것이다. 통합 규칙들이 존재할 때는 이와는 달리 증가(augmentation), 축소(minimization), 축약(reduction)의 세 가지의 기본 연산자를 이용하여 질의를 최소화한다. 증가 연산자는 잘 알려진 추적 절차를 이용하고, 축소 연산자는 이체 동형(homomorphism)

기법을 기반으로 이루어진다. 알고리즘은 처음에 통합 규칙을 사용하여 ACIM 알고리즘으로 트리 패턴을 증가시킨 후에 CIM 알고리즘을 이용하여 유일한 최소화된 동등 질의를 찾아낸다. Amer Yahia et al.은 이를 CDM 알고리즘이라 명명했다.

3. 선형 XPath 질의 축약

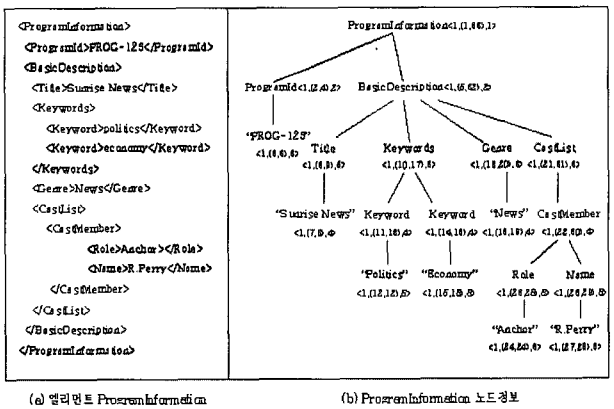
3.1 질의 축약 기법의 개념 및 효과

본 논문이 제안하는 XML 질의 최소화라는 개념 및 이것의 효과를 먼저 설명하고자 한다. (그림 2)-(a)와 같은 구조의 'ProgramInformation' 엘리먼트가 다수 반복되어 포함되어 있는 XML 문서를 데이터베이스화 한다고 가정하자. (그림 2)-(b)는 노드 번호 매기기 방식에 의한 (그림 2)-(a)의 각 엘리먼트의 노드정보를 나타내는 트리이다. 이에 대해 사용자 질의가 예를 들어 "모든 Program에 대해 출연하는 사람의 이름을 검색하라"였을 때, 이에 대한 XPath 질의문은 아래와 같이 다양하게 표현 될 수 있다.

```
//ProgramInformation/BasicDescription/CastList/
  CastMember/Name                                    (1)

//BasicDescription//CastMember/Name                  (2)

//CastList//Name                                     (3)

//Name                                               (4)
```

이러한 질의문들이 시스템의 질의 처리기 입장에서는 전혀 다른 질의 수행 계획을 만들 수가 있다. 즉, 질의문 (1)은 연속된 노드 쌍들이 부모 자식 관계를 이루고 있으므로 4번의 구조적 조인이 필요한 반면, 질의문 (4)는 단 한번의 구조적 조인도 필요하지 않고 동일한 내용의 질의를 수행할 수 있다.

이 예에서 보다시피 사용자가 명세 하는 XML 질의는 임의적인 것이어서, 질의 처리기는 구조적 조인의 횟수를 가능하면 줄일 수 있도록 하는 질의 축약 과정이 필요하게 된다.



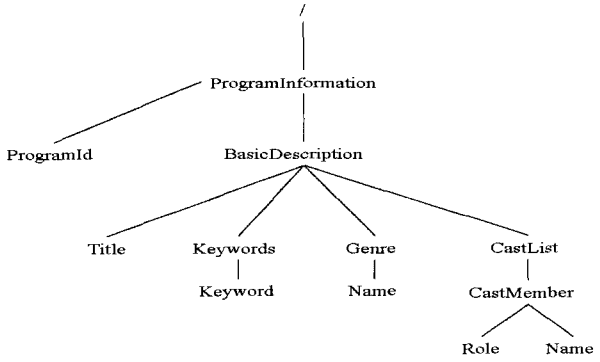
(그림 2) XML 문서와 노드 번호 매기기의 예

3.2 XPath와 XIP 트리 사이의 동등클래스 개념 정의

사용자의 트리구조 XML 질의에 대해 동등한 축약된 XML 질의를 생성하는 문제를 풀기 위해서 여기에서는 동등 클래스(equivalence class) 개념을 도입한다. 즉, 동일한 동등 클래스에 속하는 XML 질의들은 그 내용이 동등하기 때문에 상호 대체 가능하다. 또한, XML 질의 간의 동등 클래스 판별을 위해서 XIP(XML Instance Path) 트리라는 구조를 정의한다.

XIP 트리는 주어진 XML 문서의 구조를 나타내는 트리로서, XML 문서가 데이터베이스에 적재될 때 생성한다. XIP 트리는 구조적으로 유사한 경로들을 통합하여 유지하고 있으며 XIP 트리의 기본 생성 규칙은 동일한 부모 노드 아래에 있는 자식 노드들은 동일한 엘리먼트 이름이 하나만 존재하게 하는 것이다. 예를 들면 (그림 2)의 XML 문서를 이용하여XIP 트리를 생성하면 (그림 3)과 같다. (그림 2)에서는 Keyword가 여러 번 나타나지만 (그림 3)에서는 한 번만 나타난다. XIP 트리가 DTD나 XML Schema와 다른 점은 XIP 트리는 동적으로 각 입력 XML 문서에 따라 개별적으로 생성되어 달라지지만 DTD와 XML Schema는 정적이며 해당 문서 그룹 전체에 대해 동일하게 정의가 된다는 것이다. 즉 두 개의 서로 다른 XML 문서가 동일한 DTD나 XML 문서를 기반으로 하더라도 생성된 XIP 트리는 서로 다를 수 있어 XIP 트리는 각 문서 별로 정의된다는 것이다. XIP 트리는 또한 strong dataguide[14]와 매우 유사하다. 주요한 차이점이라면 XML 문서의 엘리먼트와 애트리뷰트 이름은 XIP 트리에서는 노드로 표현되며 strong dataguide에서는 간선으로 표현된다. XIP 트리가 각각의 개별적인 XML 문서들의 구조를 정확하게 표현할 수 있으므로 최적화 작업에 좀더 유용하게 사용될 수 있는 반면 dataguide는 XML 문서들의 일반적인 구조에 대한 그림을 제시할 수 있어 주로 질의 구성이나 수행시에 유용할 수 있다. XIP 트리는 XML 문서에 따라서 크기가 변동이 있을 수 있지만, 일반적으로 그 크기는 원래의 XML문서에 비해서 매우 작기 때문에 메모리에 유지하는 것이 가능하다.

주어진 XML 문서가 데이터베이스화 되고 XIP 트리가 생성되면, 선형구조의 XML 정규식 질의에 대한 동등 클래스는 다음과 정의될 수 있다[15].



(그림 3) XIP 트리의 예

정의 1. 완전 경로식(Complete path expression)

경로식 C가 '/'축으로 시작하는 절대 경로식이면서 '//'축을 포함하지 않으면 완전 경로식이다. □

정의 2. 일치하는 완전 경로식(Matching complete path expression)

임의의 경로식 X('//'축 포함 가능)가 XIP트리에서 나타내는 경로중의 하나로서 완전 경로식 C가 나타내는 경로와 일치하면, C는 X의 일치하는 완전 경로식이다. □

정의 3. 확장(Expand)

Expand 함수는 주어진 정규식 질의에 대하여 모든 일치하는 완전 경로식으로 이루어진 집합을 반환한다. 임의의 정규식 질의 X에 대하여, Expand(X)={C1,C2,…,Ck} (단, Ci(1<=i<=k)는 X의 일치하는 완전 경로식임). 즉, X의 모든 일치하는 완전 경로식은 Expand(X)의 원소이다. □

정의 4. 동등 클래스(Equivalence class)

정규식 질의 집합, Xset={X1,X2,…,Xm}(단, m>=1)이 동등 클래스일 필요충분조건은 Xset에 있는 모든 Xi(1 <= i <= m)에 대해서 Expand(Xi)=Cset인 완전 경로식 집합 Cset={C1,C2,…,Cn} (where n>=1)이 존재한다는 것이다. 즉, Xset에 있는 모든 Xi(1<=i<=m)에 대해서 Expand(X1)=Expand(X2)=...=Expand(Xi)=...=Expand(Xm)=Cset 이다. □

어떤 정규식 질의에 대해서 그것의 동등 클래스 집합을 찾는 것은 매우 복잡한 문제이다. 여기에서는 우선 선형적으로 표현된 정규식 XML 질의에 대한 효율적인 질의 축약 알고리즘을 설명하고, 이를 확장하여 트리 구조의 XML질의에 대한 질의 축약 알고리즘을 개발하고자 한다.

3.3 XPath 질의 축약 기법

질의 축약의 기본 아이디어는 축약한 결과 정규식 표현이 원래의 정규식 표현과 동등 클래스에 속할 때, 부모-자식 관계는 조상-자손 관계로 대체될 수 있다는 것이다. 예를 들어 경로식 "//ProgramInformation/BasicDescription/CastList/CastMember/Name"이 있을 때, 이것은 4개의 연속적인 부모-자식의 노드의 쌍으로 나타낼 수 있다. {(ProgramInformation, BasicDescription), (BasicDescription, CastList), (CastList, CastMember), (CastMember, Name)}. 이 부모-자식 노드 쌍은 조상-자손 관계를 이용하여 (root, CastMember)와 (CastMember, Name)으로 나타낼 수 있다. 즉 "//CastMember/Name"로 교체될 수 있다.

주어진 정규식 표현에서, 동등 클래스에 속한 모든 가능한 표현 가운데 가장 짧은 경로 를 찾기 위해 각 표현을 다 평가하기엔 그 수가 너무 많다. 질의 축약 과정을 단순화하기 위해 우리는 2가지 방식으로 접근한다. 하나는 욕심쟁이(greedy) 알고리즘을 이용하는 것이다. 알고리즘은 주어진 정규식 표현의 각 노드를 왼쪽에서 오른쪽으로, 즉, XIP 트리에서의 하향식(Top-Down) 순서로 연속적으로 탐색하면서 그 노드를 제거해야할지 결정한다. 그 노드가 제거 가능한 노드라고 판단되면 노드 제거 후, 그 선행 축은 '//'로 대체된다. 노드는 결과 경로의 표현이 노드를 제거하고 난 후에도 기존의 동등 클래스에 여전히 속해야 제거할 수 있다. 만일 임의의 노드를 정규 표현으로부터 제거하면, 결과는 최초의 경로 표현의 동등 클래스의 멤버가 아닌 경로가 될 수 있다.

다른 한 가지 접근 방법은 이와는 반대로 트리의 리프 노드부터, 즉 질의문의 오른쪽부터 처리하는 상향식(Bottom-up) 방식이다. 앞에서 제안된 욕심쟁이 알고리즘은 몇 가지 문제점을 가지고 있다. 하나는 Expand() 함수가 매번 원래의 경로식에서 하나의 노드를 호출할 때마다 필요하다는 것이다. 이 같은 이유 때문에 경로 트리가 복잡할 경우 시스템이 Expand() 함수를 호출하는 횟수가 많아져 알고리즘의 수행 시간이 현저하게 늘어나게 되어 알고리즘이 매우 비효율적 수행을 보이게 된다. 다른 문제는 하향식(Top-Down) 알고리즘은 왼쪽에서 오른쪽으로 진행되는 방식으로 노드를 검색하기 때문에 가장 효율적인 최소 경로를 찾을 수 없을지도 모른다는 사실이다. XIP 트리의 리프에 가까운 쪽의 노드(즉, 경로식의 오른쪽의 노드)부터 제거하는 것은 루트 가까이의 노드를 제거하는 것보다 더 효과적이다. 왜냐하면 대개 어떤 노드가 XIP 트리의 리프에 가까우면 XML문서에서 그 노드가 반복되는 경우가 더 많아지기 때문이다. 하향식(Top-Down) 알고리즘은 경로식의 왼쪽 노드에 높은 우선순위를 부여하고 이부터 제거하려고 하기 때문에, 마땅히 제거되어야하는 오른쪽의 노드들이 제거되지 않을 수도 있다.

선행 연구[15]에서는 상향식(Bottom-Up) 접근을 통해 이러한 문제점을 해결하고 더 나은 알고리즘을 제안한다. 경로 트리의 노드 이름을 해쉬 키로 가지고 있는 해쉬 테이블이 있다고 가정해보자. 해쉬 테이블의 각 버킷은 경로 트리에 있는 각각의 노드 이름에 대해 노드의 ID를 포함하고 있다. 알고리즘은 기존 경로식의 가장 오른쪽 노드의 이름과 동일한 이름을 갖는 경로 트리의 노드를 찾기 위해 해쉬 테이블을 검색한다. 찾은 노드는 임시로 저장된다.

예를 들어, "//ProgramInformation/BasicDescription/CastList/CastMember/Name"이라는 경로가 있을 때, Name과 동일한 이름을 갖는 노드 ID들을 임시로 저장한다. 알고리즘 초기에 생성되는 가장 짧은 경로는 "Name"이다. 기존 경로식에 있는 각각의 노드는 오른쪽에서 왼쪽으로 가는 순서로 고려되며, 다음의 단계를 반복한다.

최초 경로식의 노드가 오른쪽에서 왼쪽으로 검색되는 동안에, 더 짧은 경로식 표현을 위해 최초 경로식의 어떤 노드들을 '//'로 바꿀 수 있는지를 XIP 트리 안에서 상향식(bottom-up) 방식으로 조사한다. 이것은 고정(anchor) 노드인지 아닌지에 대한 식별을 필요로 한다. 고정 노드란 경로식을 축약한 이후에도 경로식에 반드시 포함되어야 하는 노드로서 고정 노드가 없으면 원래 축약 이전의 경로식과는 다른 의미를 갖는 경로식이 되어버린다. 즉 경로식을 축약할때에 제거가 불가능한 노드이다. 어떤 노드를 최초 경로식의 왼쪽에서 오른쪽으로 찾아갈 때, 이와 마찬가지로 XIP 트리에서도 상응하는 경로로 따라갈 수 있어야 하며, 이 경로를 따라 처리한다. 만약 지금까지 구성된 경로 중에 그

노드를 제거했을 경우 최초 경로식의 경로에 동등하게 되지 못할 경우, 그러한 노드를 고정 노드로 나타낸다. 이 경우, 고정 노드는 축약한 경로 안에 포함되어야만 한다. 그와는 달리 해당 노드를 제거한 지금까지 구성된 모든 경로가 최초의 경로식과 동등하게 될 경우, 원래의 경로식에서 그 노드를 제거하고 '//'로 대체할 수 있다. 이때 '/'와 '//'같이 타입이 다른 축 사이를 움직일 때 조금 복잡해진다. 만일 '/'축이면, 각 경로를 위해 XIP 트리에서 단지 부모 노드만을 체크하는 것으로 충분하다. 그러나 '//'축일 경우, XIP 트리의 각 경로는 부합하는 노드의 모든 발생 경우를 찾기 위해 루트 노드까지 조사해야하고, 고려되는 새 경로를 위해 그것들을 시작점으로 만들어야한다. 만일 어떤 경로가 루트까지 올라갈 수는 있으나 상응하는 노드가 발견되지 않으면, 그 노드는 고정 노드이고 축약한 결과 경로식에서 제거될 수 없다. XIP 트리를 조사하는 가운데 상응되는 노드가 확인되면 그 노드는 '//'로 바뀔 수 있다.

4. 트리 구조의 XPath 질의 축약 기법

4.1 트리 구조의 XPath 질의 축약을 위한 동등클래스 정의

XPath에서는 노드를 트리 구조로 인식하기 때문에 위와 같은 선형의 질의 축약 알고리즘으로는 충분치 않다. 조건절이 붙는 질의에서의 처리는 질의 자체가 트리 구조로 형성되는 질의 트리가 형성되는데 이러한 경우의 처리는 선형 알고리즘만으로는 부족하기 때문이다. 이러한 문제점을 해결하기 위해 본 논문에서는 분기 질의를 처리하는 알고리즘을 제안한다.

트리 구조의 질의, 즉 분기 노드가 포함된 경로식의 질의 축약 알고리즘은 이전부터 선형 경로를 줄이기 위한 경로 축약 알고리즘으로부터 발전되어왔다. 분기 노드가 포함된 질의 축약의 아이디어는 다음의 두 가지 과정으로 진행된다. 1단계 과정은 트리 구조의 질의가 확인되면 질의 가운데 중복 노드가 존재하는지 확인하고 이 때 중복 노드가 존재하는 경우, 중복 노드를 삭제하고 하나의 노드로 통합하여 질의 트리의 모양을 변형시키는 과정이다. 만약 중복 노드가 존재하지 않는 경우에는 1단계 과정은 생략된다. 그리고 2단계 과정은 이렇게 처리된 질의 트리를 가지고 분기 노드를 중심으로 여러 개의 하위 경로식으로 나누어 각각을 선형 질의 축약 알고리즘을 이용하여 축약한 후 분기 노드를 중심으로 다시 합치는 과정이다.

중복 노드 통합을 위해서는 분기 노드가 발견되었을 때, 그 발견된 분기 노드를 삭제했을 때의 경로식이 원래의 경로식과 동등 클래스에 속한다면 분기 노드를 삭제하고 하위 경로를 통합할 수 있다는 가정이 필요하다. 이를 위해서는 앞의 선형 질의를 처리하기 위해 정의했던 동등 클래스 정의를 분기 노드가 포함된 경로식 질의 즉, 트리 구조의 질의 처리를 위해 재정의하는 과정이 필요하다. 분기 질의에 대하여 재정의된 동등 클래스는 다음과 같다.

정의 3(Tree). 일치하는 완전 트리(Matching complete tree)

임의의 질의 트리 X('//'축 포함 가능, 분기 노드 포함)가 XIP트리에서 나타내는 완전 경로식으로만 이루어진 트리 T와 일치하면, T는 X의 일치하는 완전 트리이다. □

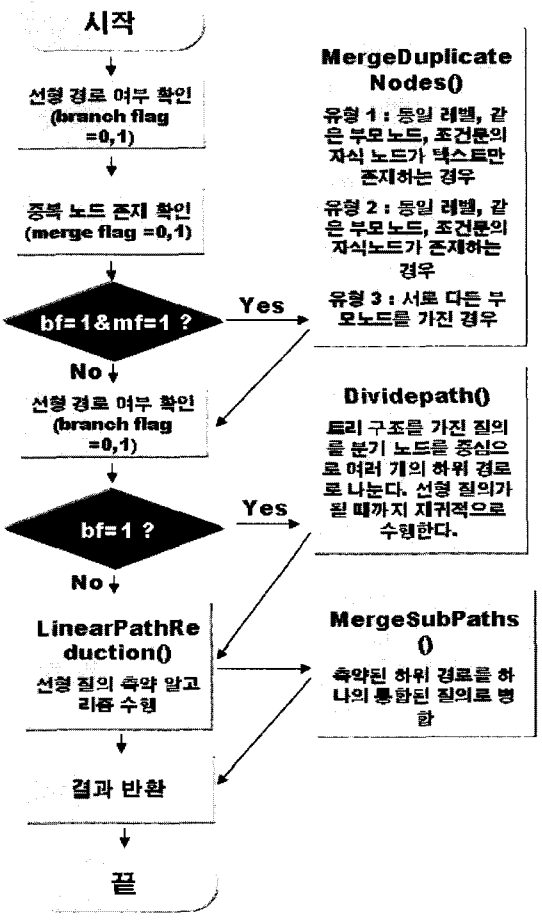
정의 4(Tree). 확장(Expand)

Expand 함수는 주어진 질의 트리에 대하여 모든 일치하는 완전 트리로 이루어진 집합을 반환한다. 임의의 정규식 질의 트리 X에 대하여, Expand(X)={T1, T2, ⋯, Tk} (단, Ti (1 <= I <= k)는 X의 일치하는 완전 트리임). 즉, X의 모든 일치하는 완전 트리는 Expand(X)의 원소이다. □

정의 5(Tree). 동등 클래스(Equivalence class)

정규식 질의 집합, Xset={X1, X2, ⋯, Xm}(단, m >= 1)이 동등 클래스일 필요충분조건은 Xset에 있는 모든 Xi(1 <= I <= m)에 대해서 Expand(Xi) = Tset인 완전 경로식 집합 Tset = {T1, T2, ⋯, Tn} (where n >= 1)이 존재한다는 것이다. 즉, Xset에 있는 모든 Xi(1<=i<=m)에 대해서 Expand(X1) = Expand(X2) = ... = Expand(Xi) = ... = Expand(Xm)=Tset 이다. □

(그림 4)은 알고리즘 흐름도이며 (그림 5)는 질의 축약 알고리즘 전체를 나타낸 것이다.



(그림 4) 트리 구조의 질의 축약 알고리즘 흐름도

BranchQueryReduction 알고리즘

```
BranchQueryReduction()
input : P = A1N1...AiNi[P1]....AnNn
        /* Ni is a branch node, Pi is a branch paths.*/
        XIPtree, HashTable

/* 트리 구조의 질의 축약 알고리즘을 위한 변수 선언*/
exist_branch_node ← NULL;
preceding_path ← NULL;
branching_node ← NULL;
branch_path[MAX] ← NULL;
branch_reduced_path[MAX] ← NULL;
result_path ← NULL;

/* 트리 구조의 질의 이고 중복 노드가 있으면 중복 노드 처리 과정을 수행 */
CheckBranchPath();
CheckDuplicateNodes();
MergeDuplicateNodes();
CheckBranchPath();
/* 중복 노드 처리 후 경로식이 트리 구조의 질의인지를 다시 확인 */
/* 경로식이 선형 질의인 경우 선형 질의 축약 알고리즘 수행*/
if exist_branch_node = FALSE then
  result_path = LinearPathReduction(P, XIPtree, HashTable);
end

/* 경로식이 트리 구조의 질의인 경우*/
else
  for each i from 1 to n do      /* 분기 노드가 아닌 경우 */
    if  CheckBranchNode() = FALSE then
      preceding_path ← preceding_path + AiNi;
    end
    else
      Dividepath();
      number_count ← count ;
      count ← 0;
      /* 각 경로는 재귀적으로 축약을 수행한다. */
      for each k from 0 to number_count   1 do
        branch_reduced_path[k]=
            BranchQueryReduction(branch_path[k], XIPtree, HashTable);
      end

      /* 분기 노드를 중심으로 병합 */
      MergeSubPaths();
      break;
    end
  end
end
return result_path ;
```

(그림 5) BranchQueryReduction 알고리즘

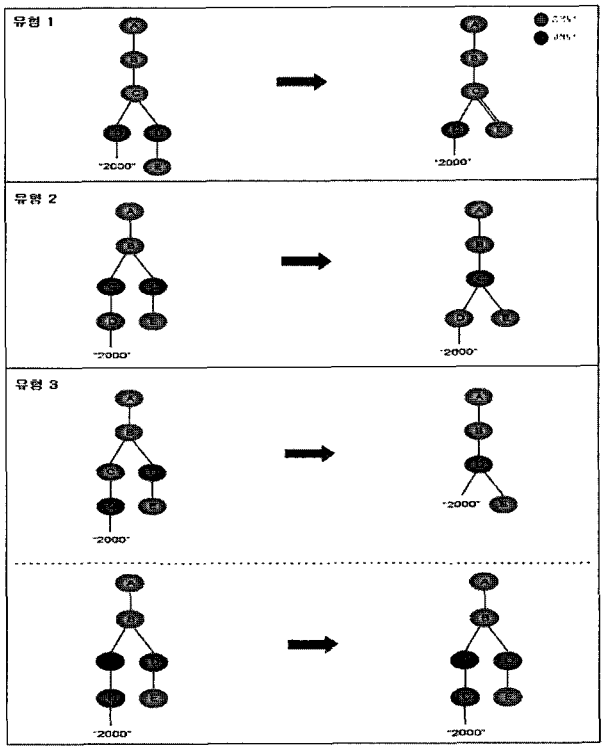
4.2 중복 노드 처리를 통한 질의 트리 변형

트리 구조를 가진 질의가 발견되면 알고리즘은 가장 먼저 이 질의에 중복 노드가 포함되어 있는지 확인한다. 중복 노드가 있다고 판명되면 중복 노드 처리를 통해 질의의 모양을 변형시키고, 중복 노드가 없으면 이 과정 없이 다음 단계의 과정을 수행한다.

중복 노드의 처리는 일단, 경로식이 입력되면 이 경로식이 분기 노드를 포함하는지, 아닌지를 판단하는 것부터 시작한다. 분기 노드가 있으면 트리 구조의 질의임을 나타내는 플래그(flag)값이 1로 결정되고, 없으면 선형 질의라는 의미로 그대로 0값을 유지하게 된다. 그 이후 중복 노드가 존재하는지를 판단하는 과정을 거친다. 이 과정에서 중복 노드가 발견되면 알고리즘은 중복 노드의 존재 여부를 의미하는 플래그(flag)가 1값을 가지게 되어 중복 노드의 존재를 알린다. 이 플래그 값은 초기에는 0값을 갖는다. 이렇게 입력된 질의를 판별하고 나서 질의가 트리 구조를 가지면서 중복 노드를 포함하고 있다고 결정되면 알고리즘은 중복 노드를 처리하는 과정을 수행한다. 중복 노드는 3가지 유형에 따라 다르게 처리된다. 3가지 유형은 (그림 6)과 같다.

유형 1. 동일 레벨, 하나의 부모 노드, 조건문의 자식 노드가 텍스트만 존재하는 경우:

동일한 레벨에 존재하고 하나의 부모 노드를 가진 중복 노드에서는 하나의 노드가 다른 중복 노드들을 대체할 수 있기 때문에 하나의 노드만을 남기고 나머지 노드를 삭제하는 방법을 사용한다. 그러나 조건문의 자식 노드가 텍스트만 존재하기 때문에 조건문의 노드만을 남기고 나머지 노드를 삭제한 후, 나머지 노드들에 연결된 자식 노드들을 '//'를 이용하여 조건문이 아닌 기존 노드에 연결시킨다. 예를 들어 /memo/header/date[year = "2000"]/year/month이라는 질의가 있다고 가정하자. 이 질의는 조건문이 존재하는 트리 구조의 질의이고 year라는 중복 노드가 포함된 질의이다. 그래서 이 질의는 중복 노드 처리 과정을 수행하게 된다. 알고리즘은 year라는 중복 노드를 인식하고 조건문에 포함된 year 말고, 기존의 경로식에 포함된 year를 삭제한 후 year에 연결된 자식 노드인 month를 date와 조상 자손 관계로 연결한다. 예제의 결과는 /memo/header/date[year="2000"] //month로 도출된다.



(그림 6) 중복 노드 처리 유형

유형 2. 동일 레벨, 하나의 부모 노드, 조건문의 자식 노드가 존재하는 경우:

유형 2는 유형 1과 같이 동일한 레벨에 존재하고 하나의 부모 노드를 가진 중복 노드이므로 이 또한 하나의 노드만을 남기고 나머지 노드를 삭제하는 방법을 사용한다. 그러나 유형 1과는 달리 조건문의 자식 노드가 존재하므로 유형 1과는 다른 방식으로 중복 노드를 처리한다. 중복 노드의 자식 노드가 존재하므로 조건문에 남겨두지 말고 기존의 경로식에 포함시켜 조건문과 경로식에 노드가 두 번 등장하지 않게 만든다. 유형 2에 대한 예제는 다음과 같이 들 수 있다. 질의 /memo/header[date/year="2000"] /date/year는 유형

2에 해당되는 예제이다. 이 때 중복 노드 date는 조건문에서 삭제되어 조건문에는 year노드만이 텍스트와 함께 존재하게 된다. 그리고 조건문 이후에 등장하던 기존 경로식의 중복 노드는 조건문 앞에 위치하여 분기 노드가 중복 노드로 교체되게 된다. 예제의 결과는 /memo/header/date[year = "2000"]/year와 같이 나타난다.

유형 3. 서로 다른 부모 노드를 가진 경우:

유형 3은 유형 1, 2와는 달리 서로 다른 부모 노드를 가진 중복 노드를 처리하는 방법이다. 이 유형을 처리하기 위해서는 우선적으로 분기 노드가 삭제 가능한지를 살펴보아야 한다. 분기 노드가 삭제 가능할 경우 분기 노드를 삭제하여 같은 부모 노드가 되면 유형 1, 2와 같이 중복 노드를 통합하고, 분기 노드가 삭제 불가능할 경우 더 이상 트리의 형태를 변화시키지 말고 다음 과정을 수행한다. 분기 노드의 삭제 가능 여부를 알기 위해서는 재 정의된 Expand() 함수를 사용한다. 분기 노드를 삭제한 후의 질의에 대해 Expand() 함수를 수행한 결과와 분기 노드를 삭제하기 이전의 질의에 대해 Expand() 함수를 수행한 결과가 같다면 이는 동등클래스에 속하는 질의로 삭제 가능하고, 두 질의에 대한 Expand() 함수 수행 결과가 다르면 분기 노드는 삭제 불가능하다. (그림 7)은 중복 노드 통합 알고리즘을 나타낸 것이다.

4.3 분할과 병합 기반의 트리 질의 축약

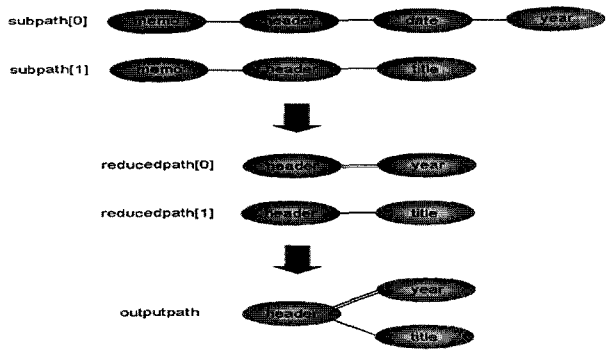
질의 트리를 변형하는 방식으로 중복 노드를 처리하고 난 후의 질의는 기본 축약 방식을 이용하여 질의를 축약한다. 이는 분할과 병합(divide-and-conquer) 방식으로 진행된다. 우선 질의를 분기 노드를 중심으로 여러 개의 하위 경로로 나누어 저장한다.

(그림 9)은 분할 알고리즘을 기술한 것이다. 알고리즘은 이렇게 분할된 각각의 경로를 선형 질의 축약 알고리즘을 이용하여 재귀적으로 축약한다. 그러나 기존의 선형 질의 축약 알고리즘과는 달리 분기 노드를 지정하여 이를 삭제하였을 경우에는 기존의 Expand() 함수 말고 재정의된 Expand() 함수로 평가될 수 있게 한다. 이렇게 축약된 각각의 경로는 분기 노드를 중심으로 다시 병합하여 하나의 경로를 형성하게 된다. (그림 10)에는 병합 알고리즘이 설명되어 있다. 이후에는 병합하여 하나로 형성된 경로를 최종 결과로 저장하여 반환한다. 예를 들어 /memo/header[date/year = "2000"]/title과 같은 질의가 있을 경우 알고리즘은 이 트리 구조의 질의

MergeDuplicateNodes 알고리즘

```
MergeDuplicateNodes()
input : P = A1N1...AnNn[P1]....AnNn
        /* Ni is a branch node, Pi is a branch path.*/
        XIPtree, HashTable

/* MergeDuplicateNodes 를 위한 선언*/
count ← 0;
same_node[MAX] ← NULL;
merge_data ← NULL;
merge_node ← NULL;

/* 중복 노드 존재 여부를 확인 */
for each i from 1 to n-1 do
  for each j from i+1 to n do
    if Ni = Nj then
      same_node[count++] ← Ni ;
    end  end  end

if same_node != NULL then
  for each i from 0 to count -1 do
    merge_data = getData(same_node[i]);
    for each j from 0 to same_node[i].length-1 do
      merge_node[j] = same_node[i].item(j);
    end

/* 유형 1 : 동일 레벨, 하나의 부모 노드, 조건문의 자식노드가 텍스트만 존재하는 경우 */
    if merge_nodes' level are same && nodes' parents are same
      && nodes are leaf nodes do
      delete(all nodes except merge_node[0]);
    end

/* 유형 2 : 동일 레벨, 하나의 부모 노드, 조건문의 자식노드가 존재하는 경우 */
    else if merge_nodes' level are same && nodes' parents are same
            && nodes aren't leaf nodes do
      delete(all nodes except merge_node[0]);
      /* move the delete node's children to merge_node[0] */
      moveChildren();
    end

/* 유형 3 : 서로 다른 부모 노드를 가진 경우 */
    else if merge_nodes' level are different && nodes' parents are different do
/* 분기 노드가 삭제 가능한지 확인, 삭제한 질의 트리의 Expand() 결과와 원래 질의 트리의 Expand()함수 결과가 동등클래스에 속하면 삭제 가능 그렇지 않으면 삭제
   불가능함*/
      if CheckEquivalClass() do
        delete(all nodes except the lowest level node);
        /* move the delete node's children to the lowest level node */
        moveChildren();
      end  end  end  end
```

(그림 7) MergeDuplicateNodes 알고리즘



(그림 8) 기본 축약 방식 수행 과정

Dividepath 알고리즘

```
Dividepath()
branching_node ← AiNi;
SetAnchorNode(); /* 고정 노드를 결정하는 함수 */
preceding_path ← preceding_path + branching_node;
for each j from 1 to number_children do
        if Pi's A1 = '//' then
                branch_path[count] ← preceding_path + Pi ;
                count ++;
        end
        else
                branch_path[count] ← preceding_path + '/' + Pi ;
                count ++;
        end  end
```

(그림 9) Dividepath 알고리즘

를 각각 /memo/header/date/year와 /memo/header/title로 분할하여 배열에 저장하고 이 각각의 경로를 축약한다. 선형 질의 축약 알고리즘은 이 두 경로를 //header//year와 //header/title로 축약하고 트리 구조의 질의 축약 알고리즘은 이를 header를 기준으로 병합하여 최종 결과로 //header[//year = "2000"]/title을 반환한다.

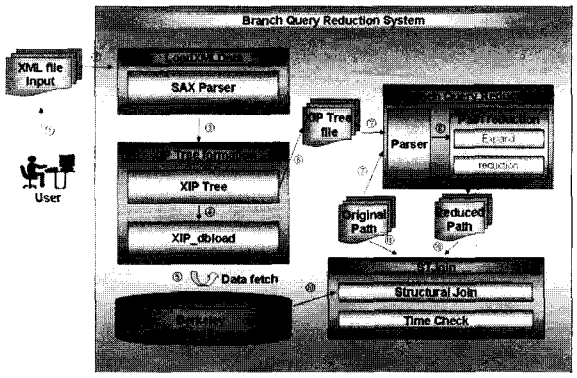
MergeSubPaths 알고리즘

```
MergeSubPaths()
for each a from 1 to branch_reduced_path[0].length do
    if branch_reduced_path[0]'s AaNa != branching_node then
        result_path ← result_path + branch_reduced_path[0]'s AaNa;
    end
    else   /* if branch path starts '//' */
        result_path ← result_path + branch_node + '[' ;
        if number_children = 2 then
            if Aa+1 = '/' then
                result_path ← result_path
                        + The rest of branch_reduced_path[0]
                                except Aa+1 + ']' ;
                result_path ← result_path + branch_reduced_path[1] ;
            end
            else  /* Aa+1 = '//' */
                result_path ← result_path + The rest of
                                        branch_reduced_path[0] + ']' ;
                result_path ← result_path + branch_reduced_path[1] ;
            end
            break;
        end
        else  /* The number of children is more than 2 */
            if Aa+1 = '/' then
                result_path ← result_path
                        + The rest of branch_reduced_path[0]
                          except Aa+1 + '|' ;
else
                result_path ← result_path + The rest of
                                        branch_reduced_path[0] + '|' ;
            end
            break;
            end
            for each b from 1 to number_children   2 do
                if branch_reduced_path[0] 's Ai = '/' then
                    result_path ← result_path
                            + branch_reduced_path[b] except Ai + '|' ;
                end
                else /* The path starts '//' */
                    result_path ← result_path + branch_reduced_path[b] + '|' ;
                end
            end
            if branch_reduced_path[b] 's Ai = '/' then
                result_path ← result_path + branch_reduced_path[b]
                                        except Ai + ']' ;
            end
            else /* The path starts '//' */
                result_path ← result_path + branch_reduced_path[b] + ']' ;
            end
            result_path ←  result_path + branch_reduced_path[b+1];
            break;
        end  end  end
```

(그림 10) MergeSubPaths 알고리즘

5. 실험 및 검증

실험을 위해 본 논문에서는 인터넷 옥션 사이트를 모델링한 XMark 벤치마크[16]에서 제공되는 xmlgen이라는 툴을 사용하여 1G 바이트 크기의 XML문서를 생성하여 노드 번호 매기기 방식에 의하여 Berkeley DB[17]에 데이터베이스화하였다. XMark벤치마크에서는 20개의 XQuery질의를 제공하고 있지만 본 실험에서는 XPath 질의들을 대상으로 실험을 해야 하므로 XQuery 내부에 사용된 XPath 질의들을 사용하였으며 이와함께 XML 문서로부터 경로식 길이에 따라 자동으로 추출하는 프로그램을 작성하여 1000여 개의 XPath 질의문들을 추출하여 동등한 최소화된 질의문들을 생성하였다. 트리 구조의 XPath 질의 축약 시스템은 세 단계를 거치도록 설계되었다. 첫 번째 과정은 XML 문서를 이용하여 XIP 트리를 형성하는 XML 적재(Loading) 과정이고, 두 번째 과정은 XML 적재 과정에서 생성된 XIP 트리를 이용하여 본격적으로 원래의 경로를 축약 경로로 만드는 질의 축약 과정이다. 그리고 마지막으로 실제 구조적 조인을 수행한 후 그 수행 시간을 측정하여 질의 축약의 성능을 확인하게 해주는 과정이 이어진다. 이 과정에서 원래의 경로의 구조적



(그림 11) 트리 구조의 질의 축약 시스템의 전체 시스템 모듈

조인 시간과 본 논문에서 제안하는 질의 축약 후 구조적 조인을 하는 시간과의 비교를 통해 본 논문에서 제안한 질의 축약 알고리즘의 효율성을 증명한다. (그림 11)은 전체 시스템 모듈을 보여준다.

5.1 트리 구조의 질의 축약 시스템의 실행 화면

본 절에서는 트리 구조의 질의 축약 시스템의 실행 화면을 소개한다. 본 시스템은 C++ 개발 환경으로 Visual Studio C++

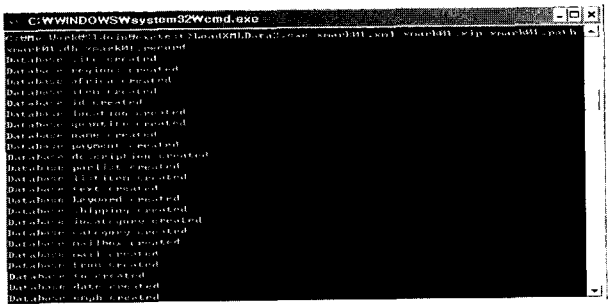
6.0을 이용하였다. 3장과 4장에 소개된 XPath 질의를 작성하고 출력된 결과를 확인하였다. 실험 샘플 파일은 xmark01.xml이고, 예제 질의는 /site/regions/africa/item[quantity = "1"]/payment이다. (그림 14)는 질의 축약 후의 결과 질의 및 질의 수행에 결린 시간을 보여주는 질의 실행 결과 화면이다.

5.2 실험 결과

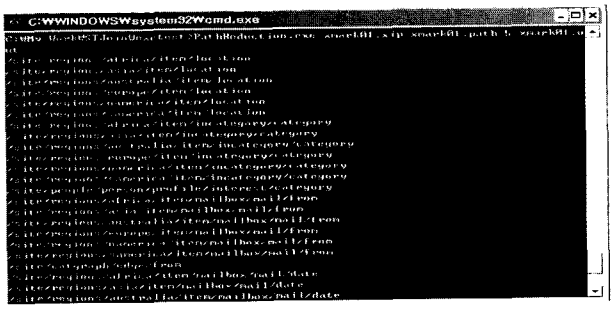
본 논문에서는 제안된 질의 축약 기법의 효용성을 2가지 실험을 통해 증명하였다. 첫째로 다양한 경로식 길이를 갖는 질의들에 대해서 원래 질의의 구조적 조인의 수와 축약된 질의의 구조적 조인의 수를 비교해보았고, 둘째로는 원래 질의를 수행하는데 걸리는 시간과 축약된 질의를 수행하는데 걸리는 응답 시간을 비교하였다. (그림 16)은 경로식 길이를 변화시키면서 원래의 질의와 축약된 질의의 구조적 조인의 수를 그래프로 나타낸 것이다. 실험에 사용된 질의

```
<?xml version="1.0" standalone="yes"?>
<site>
<regions>
<africa>
<item id="item0">
<location>United States</location>
<quantity>5</quantity>
<name>duteous nine eighteen </name>
<payment>Creditcard</payment>
<description>
<parlist>
<listitem>
<text>
page rous lady idle authority capt professes stabs monster petition heave humbly removes re
</text>
</listitem>
<listitem>
<text>
shepherd noble supposed dotage humble servilius bitch theirs venus dismal wounds gum me
</text>
</listitem>
</parlist>
</description>
<shipping>Will ship internationally, See description for charges</shipping>
<incategory category="category53"/>
<incategory category="category41"/>
<incategory category="category97"/>
<incategory category="category78"/>
<incategory category="category1"/>
<mailbox>
<mail>
<from>Libero Rive mailto:Rive@hitachi.com</from>
<to>Benedikte Glew mailto:Glew@sds.no</to>
```

(그림 12) 실험에 사용한 xmark01.xml



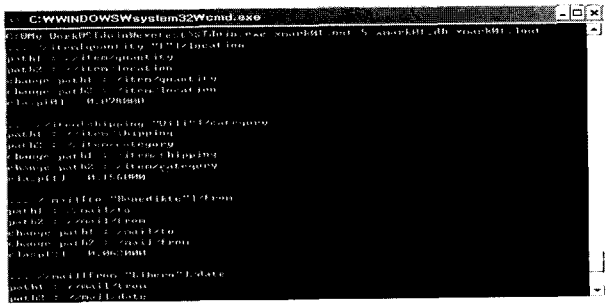
(그림 13) LoadXMLData 단계 수행 과정



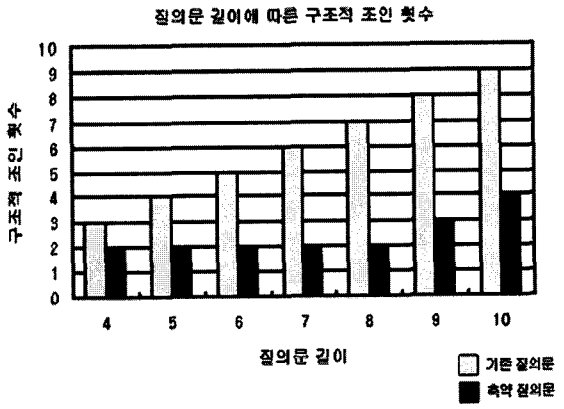
(그림 14) PathReduction 단계 수행 과정

문은 4부터 10까지의 길이를 가지고 있고, 이에 대한 원래 질의문의 구조적 조인 수는 (질의문의 길이-1)로 계산하였다. 실험 결과 질의문 길이가 길어질수록 구조적 조인 수도 늘어나는 것을 볼 수 있다. 그러나 축약 후의 질의문의 길이는 질의문의 길이에 따라 항상 변하는 것은 아니라는 사실도 확인할 수 있었다. 기존 경로식 질의문에 비해 축약된 질의문의 구조적 조인 수는 대략 1/3로 줄어든다는 사실도 (그림 16)를 통해 확인 할 수 있다.

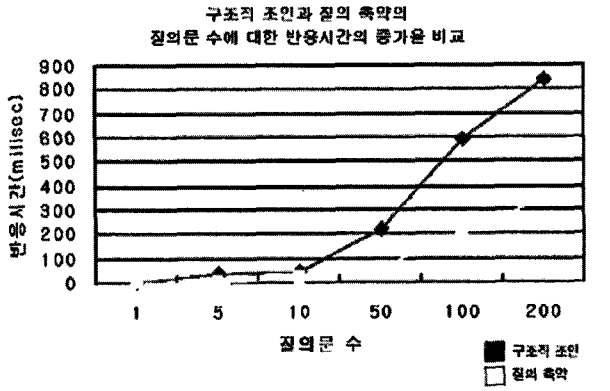
(그림 17)부터 (그림 19)까지는 질의문의 개수를 변화시키며 질의 수행에 대한 응답 시간을 비교한 그래프들을 나타낸 것이다. (그림 17)은 구조적 조인 수행시간에 비해 질의 축약에 걸리는 시간의 증가율이 더 낮음을 보여준다. 그러므로 적은 축약 시간으로 효율적으로 구조적 조인 수행시간을 낮춤을 증명한다. 이 때 측정한 축약 시간은 파일에 포함된 질의문 전체가 축약되는 총 축약시간임을 밝혀둔다. (그림 18)는 질의문 개수를 변화시키며 응답 시간을 나타낸 그래프로 구조적 조인에 걸리는 시간과 질의 축약에 걸리는 시간을 따로 나타낸다. 마지막으로 (그림 19)역시 질의문 개수에 따른 응답 시간을 나타내는 그래프로 기존 질의문은 구조적 조인을 수행한 시간만을 포함시켰고, 축약 질의문에는 축약 시간과 구조적 조인을 수행하는데 걸리는 시간을 모두 응답 시간에 포함시켰다. 그래프를 살펴보면 질의문 수가 늘어나면 늘어날수록 그에 대한 응답 시간도 늘어난다는 사실을 확인할 수 있다. 또한 <표 1>을 보면 축약을 수행하는데 걸리는 시간은 구조적 조인 수행 시간에 비해 매우 적게 걸림을 알 수 있다. 이와 같은 실험 결과로 미루어



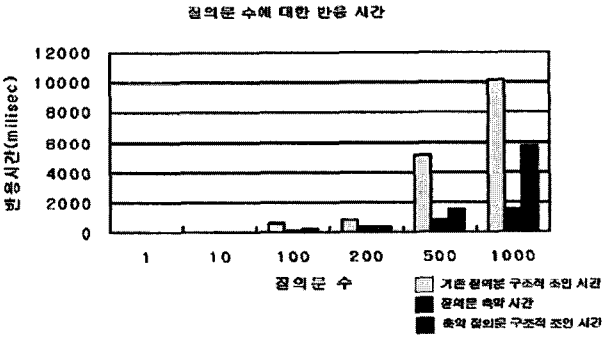
(그림 15) STJoin 단계 수행 과정



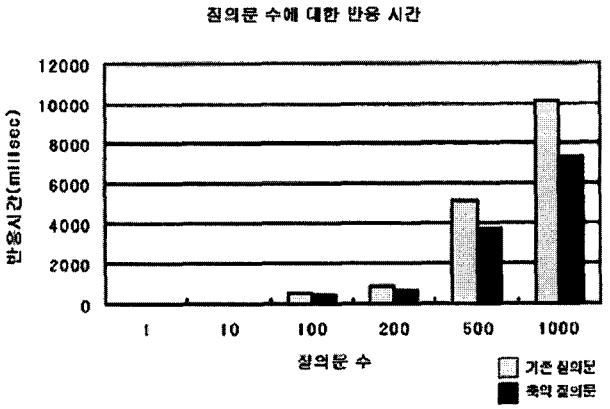
(그림 16) 질의문 길이의 변화에 따른 기존 및 축약 질의의 구조적 조인 횟수 비교



(그림 17) 질의문 수에 따른 구조적 조인 시간과 질의 축약 시간의 증가율 비교



(그림 18) 질의문 수에 따른 기존 질의 및 축약 질의의 응답 시간 비교 (축약시간과 조인시간을 분리하여 표현)



(그림 19) 질의문 수에 따른 기존 질의 및 축약 질의의 응답 시간 비교 (축약시간과 조인시간을 통합하여 표현)

볼 때 구조적 조인의 횟수가 응답 시간에 큰 영향을 미치며, 시스템의 질의 수행 성능을 향상시키기 위해서는 축약에 따른 약간의 오버헤드로 구조적 조인의 수를 줄이는 것이 유리함을 알 수 있다.

다음은 타 기법과 본 논문에서 제안한 기법의 차이점을 비교하여 논하고자 한다. 비교 대상이 되는 타 시스템은 앞장의 관련 연구에서 언급된 논리 기반의 XPath 질의 최적화 기법과 구조 기반의 XPath 최적화 기법이다. 대상이 되는 두 기법 모두 본 기법과는 달리 일정한 규칙 집합을 형성시킨 후 그 규칙에 적합하도록 질의를 최적화 시킨다. 이러한 경우 축약된 질의의 의미를 정확하게 유지할 수 있다는 장점을 가지고 있으나 3가지의 약점도 지니게 된다. 우선, 첫째로 일반적으로 적용 가능한 규칙을 생성하기 위한 복잡한 추가적인 준비 과정이 필요하다는 점이다. 이는 두 최적화 기법이 본 논문에서 제안하는 기법에 비해 처리 과정이 복잡하다는 단점을 가지게 한다. 둘째, 대량의 규칙들의 집합을 유지하고 적용하기 위한 추가적인 메모리 소모가 필요하다는 점이다. 반면에 본 기법에서는 XIP 트리만을 유지하면 된다. 마지막으로 규칙 집합에만 의거하여 축약이 이루어지므로 결과 경로식이 최적화 된 상태임을 보장하지 못한다. 본 논문의 기법에서는 축약 과정의 소모 시간이 매우 적으며 Expand() 함수를 통한 동등 클래스 비교를 통해 최적을 보장하고 있다. 또한 조건문이 포함되는 질의에 대해서도 처리할 수 있다는 장점도 가진다.

〈표 1〉 질의문 수에 따른 기존 및 축약 질의의 응답 시간

질의문 수	기존 경로식 응답시간	축약 경로식 응답시간	
		축약 수행 시간	구조적 조인 수행 시간
1	0	0	0
5	32	6.3	21.2
10	45	15	20.4
50	220	67	113.31
100	589	159	233.19
200	844	313	333.72
500	5157	773	2941
1000	10128	1570	5719

(단위: 밀리초)

6. 결 론

XML이 산업의 많은 분야에서 사용됨에 따라 대용량 XML 데이터를 효율적으로 저장하고 검색하는 기술은 점점 더 중요한 기술로 인식되고 있다. 본 논문에서는 트리 패턴의 XML 질의 처리시의 중첩된 다수의 구조적 조인의 횟수를 최소화하기 위한 XML 질의 축약 알고리즘을 제안하였다. 본 논문은 기존의 축약 방법에서 한 단계 발전한 조건문을 포함하는 질의에 대해 분할 및 병합 방법을 이용하며 개별적인 XML 문서 구조를 반영한 XIP 트리를 사용하여 상향식으로 축약을 효율적으로 수행하도록 제안하였고 좀 더 우수한 축약 결과를 제공하기 위해 중복 노드 제거를 통한 트리 변형을 시도하였다. 제안한 알고리즘의 효율성을 트리 구조의 XPath 질의 축약 시스템을 구현하여 측정하였고, 이를 통해 질의 처리 시간을 평균적으로 1/3정도 줄일 수 있음을 증명하였다. 향후에는 완전한 트리 구조의 XPath 질의 축약을 XQuery 질의 처리와 관련지어 연구하는 것이 필요할 것으로 보이며 복잡한 조건문을 갖는 XPath 질의 최적화에 대한 연구도 매우 흥미로운 주제가 될 것으로 본다.

참 고 문 헌

[1] Quanzhong Li and Bonki Moon. Indexing and querying XML data for regular path expressions. In Proc. of the 27th VLDB conference, Rome, Italy, Sep. 2001

[2] Chun Zhang, Jeffrey F. Naughton, Qiong Luo, and David J. DeWitt, and Guy M. Lohman. On supporting containment queries in relational database management systems. In Proc. of 2001 ACM-SIGMOD conference, Santa Barbara, CA, USA, May. 2001

[3] Divesh Srivastava, Shurug Al Khalifa, H.V. Jagadish, Nick Koudas, Jinesh M. Patel, and Yuquing Wu. Structural joins: A primitive for efficient XML query pattern matching. In Proc. of the 2002 IEEE conference on Data Engineering, San Jose, USA, Feb. 2002

[4] Haifeng Jiang, Hongjun Lu, Wei Wang, and Beng Chin Ooi, XR-Tree: Indexing XML Data for Efficient Structural Joins. In Proc. of the 2003 IEEE conference on Data Engineering, page 253-263, Bangalore, India, March 2003

[5] Hanyu Li, Mong Li Lee, Wynne Hsu, and Chao Chen, An Evaluation of XML Indexes for Structural Join. SIGMOD Record 33(3), pages 28-33, 2004

[6] Yuqing Wu, Jignesh M. Pastel, and H.V. Jagadish, Structural Join Order Selection for XML Query Optimization. In Proc. of the 2003 IEEE conference on Data Engineering, page 443-454, Athens, Bangalore, India, March 2003

[7] Shu-Yao Chien, Zografoula Vagena, Donghui Zhang, Vassilis J. Tsotras, Carlo Zaniolo, Efficient Structural Joins on Indexed XML Documents, Proceedings of the 28th VLDB Conference, Hong Kong, China, 2002.

[8] H. Jiang, H. Lu, W. Wang, and B. C. Ooi, XR-Tree: Indexing XML Data for Efficient Structural Join, Proc. of ICDE, India, 2003.

[9] Bingsheng He, Qiong Luo, Byron Choi, Adaptive Index Utilization in Memory Resident Structural Joins, IEEE Transactions on Knowledge and Data Engineering, Vol. 19, No. 6, pp.772-788, June 2007.

[10] Pierre Geneves, Jean-Yves Vion-Dury, Logic-based XPath optimization, Document Engineering Proceedings of the 2004 ACM symposium on Document engineering Pages 211-219, 2004

[11] April Kwong, Michael Gertz, Schema based Optimization of XPath Expressions, Technical report, Univ. of California dept. of Computer Science, 2001.

[12] C. Y. Chan, P. Felber, M. Garofalakis, R. Rastogi, Efficient Filtering of XML Documents with XPath Expressions, The VLDB Journal, Vol.11, pp 354-379, Dec. 2002.

[13] Sihem Amer-Yahia, SungRan Cho, Laks V. S. Lakshmanan, Divesh Srivastava, Tree Pattern Query Minimization, The VLDB Journal No.11 page 315-331, 2002.

[14] Roy Goldman and Jennifer Widom. Dataguides: Enabling query formulation and optimization in semistructured databases. In Proc. of the 23rd VLDB conference, pages 436-445, Athens, Greece, Aug. 1997.

[15] Hyoseop Shin, Minsoo Lee, An Efficient Branch Query Rewriting Algorithm for XML Query Optimization, 4th International Conference on Ontologies, DataBases, and Applications of Semantics (ODBASE 2005), LNCS 3761, Springer-Verlag, pp. 1629-1639, Agia Napa, Cyprus, Oct. 31 - Nov. 4, 2005.

[16] Albrecht Schmidt, Florian Waas, Martin L. Kersten, Michael J. Carey, Ioana Manolescu, Ralph Busse. XMark: A Benchmark for XML Data Management. In Proc. of the 28th VLDB conference, page 974-985, Hong Kong, China, Aug. 2002

[17] Sleepycat Software Inc., http://www.sleepycat.com

이 민 수

e-mail : mlee@ewha.ac.kr

1992년 서울대학교 컴퓨터공학과(학사)

1995년 서울대학교 대학원 컴퓨터공학과 (공학석사)

1995년~1996년 LG전자 미디어통신연구소 연구원

2000년 University of Florida 컴퓨터공학과(공학박사)

2000년~2002년 미국 Oracle Corporation, Senior Member of Technical Staff

2002년~현 재 이화여자대학교 컴퓨터학과 조교수

관심분야 : 데이터웨어하우스, XML, 지식기반 시스템, 웹 데이터베이스

김 윤 미

e-mail : cherish11@ewhain.net

2005년 이화여자대학교 컴퓨터학과(학사)

2007년 이화여자대학교 대학원 컴퓨터학과 (공학석사)

2006년~현 재 하나은행

관심분야 : XML, XPath, 데이터웨어하우스, 데이터 마이닝

송 수 경

e-mail : happymint@ewhain.net

2006년 이화여자대학교 컴퓨터학과(학사)

2006년~현 재 이화여자대학교 대학원 컴퓨터학과(석사과정)

관심분야 : XML 데이터베이스, 데이터마이닝, 임베디드 DBMS