XML 구조 문맥을 사용한 효율적인 경로 표현식 조인 알고리즘

김 학 수[†] · 신 영 재[††] · 황 진 호[††] · 이 승 미[†††] · 손 진 현[††††]

요 약

XML 데이터 검색을 위한 표준 질의 언어로서 XQuery와 XPath가 W3C에 의해 표준으로 제정되었다. XQuery와 XPath를 보편적으로 사용함에 따라, 최근 연구는 방대한 XML 데이터베이스에서 XPath 경로 표현식에 대한 효율적인 질의 처리를 위한 데이터 구조 및 알고리즘 개발에 초점을 두고 있다. 최근에, XPath 경로 표현식을 처리할 때 XML 엘리먼트 사이의 구조적 관계(조상-자손, 부모-자식)를 결정하는 구조적 조인의 개념은 중요한 XPath 프로세싱 기법중의 하나가 되었다. 그러나 XPath 질의 처리에서 자주 발생하는 구조적 조인들은 높은 비용을 요구한다.

본 논문에서, 우리는 XPath 질의들을 효율적으로 처리하기 위해 제안한 구조적 인덱스(SI) 기반의 새로운 구조적 조인 알고리즘(SISJ)을 제안한다. 실험 결과에서는 이전의 알고리즘보다 근소하게 더 효율적인 성능을 보여 준다. 그러나 재귀성이 높은 문서에 대해서는 제안기법의 가지치기 특성으로 인해 약 30% 이상의 성능향상을 보였다.

키워드 : XPath, 구조적 조인 알고리즘, 경로 표현식

An Efficient Path Expression Join Algorithm Using XML Structure Context

Hak Soo Kim[†] · Youngjae Shin[††] · Jin-Ho Hwang[††] · Seung Mi Lee[†††] · Jin Hyun Son[††††]

ABSTRACT

As a standard query language to search XML data, XQuery and XPath were proposed by W3C. By widely using XQuery and XPath languages, recent researches focus on the development of query processing algorithm and data structure for efficiently processing XML query with the enormous XML database system. Recently, when processing XML path expressions, the concept of the structural join which may determine the structural relationship between XML elements, e.g., ancestor-descendant or parent-child, has been one of the dominant XPath processing mechanisms. However, structural joins which frequently occur in XPath query processing require high cost.

In this paper, we propose a new structural join algorithm, called SISJ, based on our structured index, called SI, in order to process XPath queries efficiently. Experimental results show that our algorithm performs marginally better than previous ones. However, in the case of high recursive documents, it performed more than 30% by the pruning feature of the proposed method.

Key Words : XPath, Structural Join Algorithm, Path Expression

1. 서 론

XML[1]은 엘리먼트(Element)라고 하는 정보컨테이너(Information Container)로 구성된다. 이러한 컨테이너들은 다른 컨테이너에 완전히 중첩될 수 있지만 각각의 컨테이너는 하나의 부모 컨테이너를 가져야만 한다. 모든 XML 문서는 루트 엘리먼트라 불려지는 하나의 최상위 컨테이너를 가진다. 이러한 컨테이너들과 그들의 구조적 관계가 데이터의 의미를 기술하는데 사용될 수 있다. 우리는 데이터 컨텐츠 뿐만 아니라 다양한 데이터 포맷을 수용하기 위해 새로운 엘리먼트들을 생성하거나 기존의 엘리먼트들을 확장할 수 있기 때문에, XML은 데이터베이스와 같은 완전히 구조화된 데이터 뿐만 아니라 웹 문서와 같은 반구조적 데이터를 표현하는데 널리 사용할 수 있다. 오늘날 XML은 데이터 저장소, 데이터 표현, 커뮤니케이션 메시지와 같은 많은 애플리케이션 영역에서 효율적으로 이용되고 있다.

XML의 사용이 증가함에 따라 XML 데이터를 질의할 수 있는 XQuery, XPath와 같은 XML 질의 언어들이 W3C[2, 3, 4]에 의해 표준으로 제정되었다. XQuery는 XPath를 기반으로 하기 때문에 XPath 스타일의 질의들을 효율적으로 처리하기 위한 몇몇의 연구가 진행되어 왔다[5, 6, 7, 8, 9, 10,

※ 본 연구는 정보통신부 및 정보통신연구진흥원의 대학 IT연구센터 육성·지원 사업(IITA-2006-C1090-0603-0031)의 연구결과로 수행되었음.
이 연구(논문)는 산업자원부 지원으로 수행하는 21세기 프론티어 연구개발사업(인간기능 생활지원 지능로봇 기술개발사업)의 일환으로 수행되었습니다.
† 준 회 원 : 한양대학교 컴퓨터공학과 박사과정
†† 준 회 원 : 한양대학교 컴퓨터공학과 석사과정
††† 정 회 원 : 한양대학교 컴퓨터공학과 BK21 AIS 사업팀 연구원
†††† 종신회원 : 한양대학교 컴퓨터공학과 조교수
논문접수 : 2007년 5월 22일, 심사완료 : 2007년 6월 28일

11, 12]. XPath 질의 처리를 위해, 하나의 XML 문서는 어떤 두 노드들 사이의 부모/자식 관계를 형성하는 간선(edge)과 노드들의 7가지 타입들로부터 구성된 트리 데이터 모델로 변환된다. XPath 질의들은 XML 트리 데이터 모델에서 경로 표현식으로 작성된다. 다음은 XPath 표현식의 한 예이다.

bib/book//author[last = 'Stevens' AND first = 'W.']

위의 경로 표현식의 의미는 대상이 되는 XML 문서에서 루트 엘리먼트는 'bib'이고 'bib'의 자식 엘리먼트로서 'book'을 가지며 'book'의 자손 엘리먼트는 'author'이다. 그리고 'author'의 자식 엘리먼트인 'last'의 값은 'Stevens', 'first'의 값은 'W.'인 경로들을 나타낸다. "/"는 부모-자식 관계, "//"는 조상 - 자손 관계를 나타낸다. "[,]"는 노드 조건을 검사하기 위해서 사용된다.

XPath 표현식을 처리하기 위해서, 트리 모델에 있는 모든 후보 노드들이 조건을 만족시키는지를 검사해야 한다. 검사해야 되는 노드들은 XML 문서의 사이즈에 비례해서 기하급수적으로 증가하기 때문에, 효율적인 XPath 질의 처리를 하기 위해서는 트리 데이터 모델을 탐색하기 위한 비용을 최소화하는 것이 중요하다. 최근 역 인덱스(inverted index)를 기반으로 하는 구조적 조인 알고리즘들의 개념이 이러한 문제를 해결하기 위해 제안되었다[5,6]. 구조적 조인 알고리즘은 복잡한 경로 표현식을 몇몇의 수직 표현식(수직 표현식은 한 쌍의 노드들 사이의 부모-자식 또는 조상-자손 관계를 가지는 원자성 표현식으로서 본 논문에서 정의하였다)으로 분해한 다음에 이러한 수직 표현식의 모든 처리결과를 조합한다. 예를 들면, 위의 질의에 대응하는 수직 표현식은 bib/book, book//author, author/last, author/first, last = 'Stevens', first = 'W.'이다. 구조적 조인 알고리즘에서 효율적인 수직 표현식 처리는 다른 어떤 것 보다 중요하다는 것을 알아두자. 수직 표현식 처리는 일반적으로 엘리먼트와 문자열 값에 인덱스, 탐색 알고리즘, XML 트리 노드들 사이의 관계를 검사하는 조인 알고리즘들이 사용된다. 본 논문은 구조적 조인 알고리즘에서 부모-자식과 조상-자손 관계들만을 고려한다. 왜냐하면 이러한 관계들은 트리 데이터 모델에서 가장 일반적이고 가장 중요한 노드 사이의 관계이기 때문이다.

이전의 구조적 조인 알고리즘들은 위의 수직 표현식으로 언급한 각각의 수직 구조적 관계를 처리할 때 모든 노드들의 쌍을 검사한다. 결과적으로 이러한 알고리즘들은 마지막 결과에 영향을 주지 않는 몇몇 노드들을 방문하는 오버헤드가 발생한다. 이러한 문제는 역 인덱스, B^+-tree, R-tree와 같은 전통적인 인덱스들을 사용함으로써 제거될 수 있다. 만약 우리가 전통적인 인덱스들을 사용해서 트리 노드들(즉, 엘리먼트, 문자열 값)을 인덱스화한다면 노드들 사이의 구조적 관계 정보를 잃어버리는 일반적인 문제가 발생한다. 다시 말해서, 우리는 부모-자식과 조상-자손의 구조적 관계를 보존할 수 있는 인덱스를 구축하길 원한다.

이러한 관점에서, 우리는 먼저 SI라고 불리는 구조적 인덱스(Structured Index)를 제안한다. SI는 원본 XML 트리 모델 안에서 가능한 한 많은 구조적 관계를 유지할 수 있도록 도와준다. 그런 다음에 효율적인 수직 표현식 처리를 지원하는 SI기반의 구조적 조인 알고리즘(SISJ : Structured Index -based Structured Join Algorithm)을 개발한다.

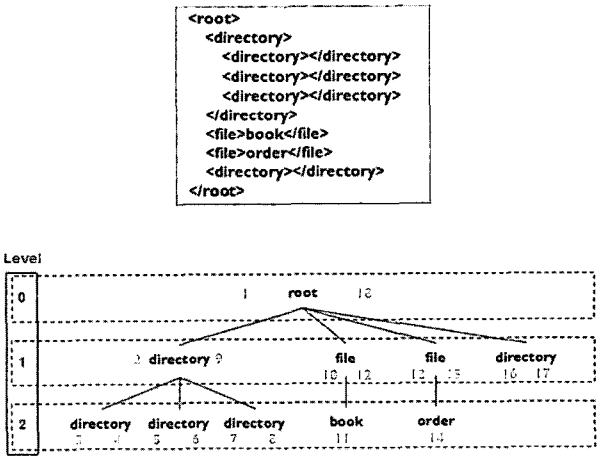
논문의 구성은 다음과 같다. 2장에서는 번호 체계(Numbering Schemes)를 이용한 역 인덱스, 구조적 조인 알고리즘에 대한 기존의 관련 연구에 대해서 알아본다. 3장에서는 기존 관련 연구의 문제정의를 통해서 SI-트리 및 SISJ 알고리즘을 제안한다. 4장에서는 알고리즘에 대한 분석을 통해 실험과 평가 내용을 제시하고 5장에서는 결론 및 향후과제로 마무리한다.

2. 관련 연구

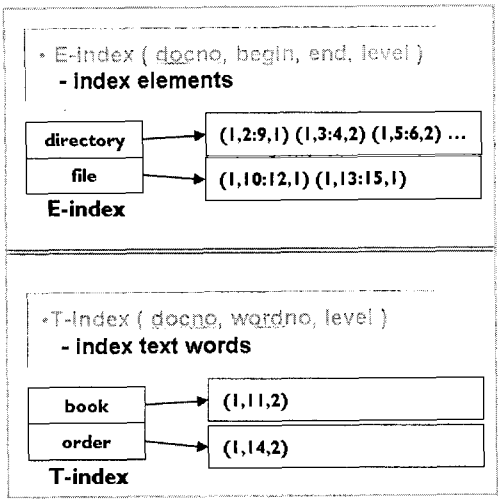
복잡한 경로 표현식을 효율적으로 처리하기 위해서 두 가지의 중요한 고려사항이 있다. (1) 트리 노드들의 임의의 쌍 중에서 구조적 관계에 참여하는 노드들을 빠르게 탐색해야 된다. (2) 경로 표현식의 구조적 관계를 만족하는 모든 노드들을 효율적으로 찾아야 한다. (1)을 해결하기 위해서 번호 체계[6]와 역 인덱스가 제안되었으며 (2)는 번호 체계기반의 역 인덱스를 이용한 구조적 알고리즘에 대한 연구가 진행되었다. 2.1절에서는 번호 체계 및 역 인덱스에 대해서 간략하게 소개를 하고 2.2절에서는 B^+-트리 기반의 구조적 조인 알고리즘을 2.3절에서는 XB-트리 기반의 TwigStackXB 알고리즘을 기술한다. 이 논문들은 구조적 조인 기반의 알고리즘들 중에서 가장 최근 기법에 해당된다.

2.1 번호 체계 및 역 인덱스

번호 체계의 목적은 서로 다른 노드들 사이의 구조적 관계를 빠르게 결정하기 위한 목적으로 개발되었다. 만약 번호 체계가 구축되어 있지 않다면 "directory"와 "file" 사이의 관계를 찾기 위해서 XML 트리 데이터 모델의 모든 노드들을 탐색해야 되는 비효율적인 문제점이 발생한다. 번호 체계는 (그림 1)과 같은 방식에 의해서 구축된다.



(그림 1) XML 문서에대한 번호 체계



(그림 2) 역 인덱스

(그림 1)에서 보는 것처럼 XML 문서는 XML 트리 데이터 모델로 변환되고 트리의 각 노드에 번호를 부여하게 된다. 부여방식은 엘리먼트, 문자열 숫자 등에 대해서 워드단위의 순차적 번호가 부여된다. 결과적으로 각 노드는 (문서번호, 시작번호, 종료번호, 레벨)의 형태로 구성된다. 예를 들면 root의 번호 체계는 (1, 1, 18, 0)이 된다. 여기에서 문서번호는 입력되는 문서에 따라 순차적으로 부여된다. 단, XML 문서에서 엘리먼트 값의 경우에는 시작번호와 종료번호가 없기 때문에 (문서번호, 워드번호, 레벨)의 형태로 구성된다. 예를 들면, 엘리먼트 "file"의 값인 "book"의 번호 체계는 (1, 11, 2)가 된다.

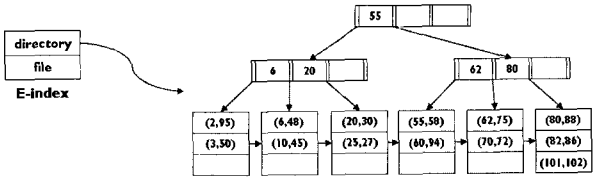
위와 같이 구성된 번호 체계를 이용하여 아래와 같은 역 인덱스를 구성할 수 있다.

(그림 2)에서 보는 것처럼 역 인덱스는 E-index와 T-index로 구분된다. E-index는 XML 문서에서 엘리먼트들을 저장하기 위해 사용되고 T-index는 값(문자열, 숫자 등)을 저장하기 위해서 사용된다. 그리고 E-index의 키에 대한 버킷은 번호 체계의 문서번호(docno), 시작번호(begin)에 의해 오름차순으로 정렬된다. T-index는 문서번호(docno), 단어번호(wordno)에 의해 오름차순으로 정렬된다.

위에서 본 것처럼, 번호 체계기반의 역 인덱스를 사용하게 되면 경로 표현식 directory//file에 대한 질의 처리를 할 때 질의 결과에 참여하는 directory와 file에 대한 노드 리스트를 XML 트리 탐색을 하지 않고 빠르게 검색할 수 있다. 그런 다음에 2.2절의 구조적 조인 알고리즘을 이용하여 directory와 file의 입력리스트 사이의 구조적 관계를 평가하여 결과를 수집한다.

2.2 B$^+$-트리 기반의 구조적 조인 알고리즘

2.1절에서 번호 체계에 기반을 둔 역 인덱스에서 키에 대한 버킷은 오름차순으로 정렬된 리스트로 구성된다. 이와 같은 리스트의 문제점은 문서의 사이즈에 비례해서 경로 표현식의 결과 값에 참여하지 않는 노드들의 순차 탐색 비용이 증가한다는 것이다. 즉, 문서의 사이즈에 비례해서 효율성이 감소된다. Shu-Yao Chien 등에 의해 제안된 B$^+$-트리 기반의 구조적 조인 알고리즘[7]은 이와 같은 문제점을 해결하는데 초점을 두고 있다. 이 논문에서 제안한 자료구조는 XML 문서에 대한 B$^+$-트리를 통해서 결과 값에 참여하지 않는 노드들을 가지치기하도록 한다. (그림 1)의 XML 문서에 대한 B$^+$-트리 구성은 아래 그림과 같다.



(그림 3) 역 인덱스에 대한 B$^+$-트리

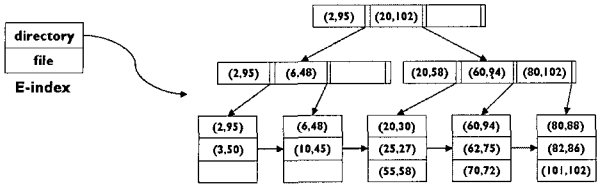
(그림 3)은 2.1절의 XML 문서에 대한 directory, file의 역 인덱스 중에서 directory 엘리먼트에 대한 B$^+$-트리가 구축된 모습을 보여주고 있다. 내부 페이지는 (키, 다음 레벨까지의 포인터)의 쌍으로 구성된다. 그리고 리프 페이지(Leaf Page)는 (노드의 시작번호, 노드의 종료번호, 다음 리프 페이지에 대한 포인터)로 이루어져있다. 문서번호와 레벨은 생략하였다.

directory//file에 대한 경로 표현식을 처리하기 위해서 먼저 directory와 file에 대한 B$^+$-트리를 역 인덱스로부터 가지고 온다. 그리고 구조적 조인 알고리즘은 두 개의 B$^+$-트리 안에 있는 노드들의 구조적 관계를 평가하기 위해서 조상 스택을 두게 된다. 리프 페이지에서는 노드들이 순차적으로 정렬되어 있기 때문에 알고리즘은 첫 번째 노드부터 순차적으로 검색을 한다. 특정 자손 노드와 조상 노드를 비교하면서 자손 노드가 조상 노드의 자손임이 결정되면 조상 스택에 삽입한다. 만약 조상-자손 관계를 만족하지 않으면 자손 노드의 조상이 되는 노드들을 B$^+$-트리를 이용하여 빠르게 찾을 수 있기 때문에 순차검색의 문제점을 해결할 수 있다. 그리고 특정 자손 노드의 조상이 더 이상 없다는 것이 판명되면 조상 스택에 있는 조상 노드들과 특정 자손 노드의 조상-자손 관계를 출력 목록에 저장한다.

2.3 XB-트리 기반의 TwigStackXB 알고리즘

XB-트리[8]는 XML 문서에 대한 B$^+$-트리처럼 입력 리스트의 크기가 커질 경우 모든 노드에 대한 순차 검색을 방지하기 위한 R-트리[9] 기반의 인덱스이다. 실제로 구축된 예제는 (그림 4)와 같다.

내부 페이지는 (시작번호, 종료번호, 자식 페이지에 대한 포인터, 부모 페이지, 부모페이지에서 현재 키의 순번)로 구성된다. *그림 4)에서는 (시작번호, 종료번호, 자식 페이지에 대한 포인터)만을 표시하였다. XB-트리는 R-트리 기반이기 때문에 키 값으로 2차원 구간을 가진다. 예를 들면, 루트 내부 페이지에 있는 (2, 95)안에는 모든 키 값 및 노드들은 이 구간 안에 포함된다는 것을 의미한다. 리프 페이지는 (시작번호, 종료번호, 다음 리프 페이지에 대한 포인터)로 구성된다. 이 페이지는 XML 문서상에서 번호 체계에 의한 노드들이 저장되는 데이터 페이지들이다. 편의상 문서번호와 레벨



(그림 4) 역 인덱스에 대한 XB-트리

은 생략하였다.

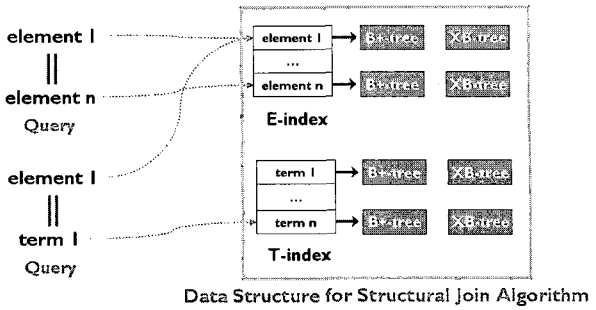
위와 같이 구축된 XB-트리를 기반으로 하는 TwigStackXB 알고리즘은 B^+-트리기반의 구조적 조인 알고리즘에서 조상-자손, 부모-자식의 구조적 관계만을 처리하는 것이 아니라 복잡한 경로 표현식에 대한 질의 처리를 목적으로 한다. 예를 들면, 제 1장 소개에서 기술된 경로 표현식인 bib/book//author[last = 'Stevens' AND first = 'W.'] 에 대해서 B^+-트리기반의 구조적 조인 알고리즘은 복잡한 경로 표현식을 조상-자손, 부모-자식 경로 표현식으로 분할한 다음 처리하지만, TwigStackXB 알고리즘은 복잡한 경로 표현식을 그대로 처리한다는 차이점이 있다. 또한, 알고리즘에서는 경로 표현식의 각 노드에 대한 스택이 모두 생성된다. 즉, bib 스택, book 스택, author 스택, last 스택, 'Stevens' 스택, first 스택, 'W.' 스택이 생성된다. 기본적으로 알고리즘은 복잡한 경로 표현식을 처리해야 되기 때문에 입력 리스트 사이의 구조적 관계를 평가하기 위해서 TwigStackXB 알고리즘은 조상 노드에 대한 스택에 자손 노드를 넣기 전에 경로 표현식의 자손 노드들을 탐색해 보고 만족할 경우에 만 스택에 넣는 방식을 취한다. 즉, 경로 표현식을 만족하는 노드들을 재귀적으로 탐색하여 결과를 수집한다. 본 논문에서는 수직 표현식을 가지는 알고리즘을 고려하기 때문에 TwigStackXB 알고리즘에 대한 자세한 사항은 생략한다[8]. 또한, 실험부분에서 TwigStackXB와 비교를 한 것은 TwigStackXB 알고리즘이 이전의 수직 표현식을 처리하는 알고리즘보다 성능이 좋기 때문이다[18].

3. 기존 시스템에 대한 분석

이번 장은 기존의 인덱스(B^+-트리, XB-트리)에 대한 결점을 분석한다. 분석된 결점을 해결하기 위한 SI-트리 인덱스를 제안하고 이를 기반으로 한 구조적 조인 알고리즘을 설계한다. 본 논문에서는 B^+-트리 기반의 구조적 알고리즘처럼 수직 표현식(조상-자손, 부모-자식)을 가지는 경로 표현식을 처리하는 알고리즘에 초점을 맞춘다. 복잡한 경로 표현식은 수직 표현식으로 분할하여 처리할 수 있기 때문에 이에 대해서는 다루지 않는다.

3.1 B^+-트리와 XB-트리의 결점

(그림 5)에서 보는 것처럼 B^+-트리와 XB-트리의 공통점은 XML 문서에서 대한 번호 체계를 통해서 같은 엘리먼트에 대한 인덱스를 만든다는 것이다. 수직 표현식을 처리하기 위해서 질의의 노드에 해당되는 입력리스트를 역 인덱스



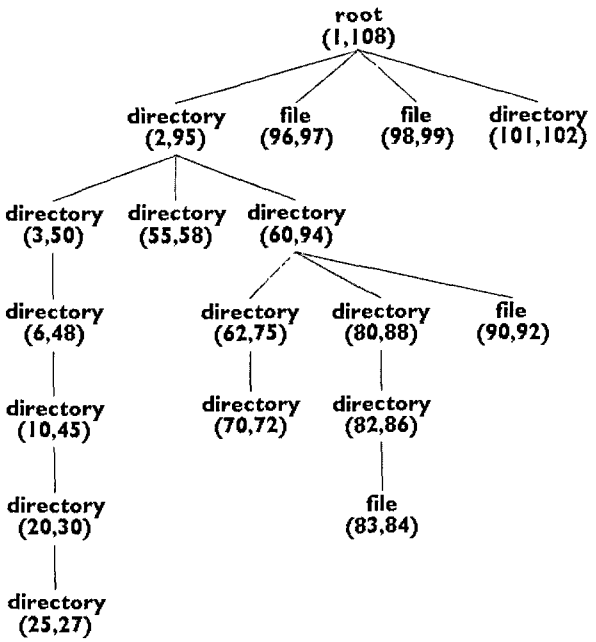
(그림 5) 수직 표현식에 대한 입력리스트의 관계

(E-index, T-index)를 통해서 빠르게 가지고 온다. 가지고 온 입력리스트는 B^+-트리 또는 XB-트리가 된다. 두 인덱스 모두 입력리스트가 길어질 경우에 모든 노드들을 순차검색을 해야 되는 문제점을 해결하는데 초점을 맞추고 있다. 여기에서 중요한 것은 XML 문서에서 같은 엘리먼트라 할지라도 엘리먼트가 재귀적으로 중첩될 경우에는 조상-자손, 부모-자식 관계를 가지고 있다는 것이다. B^+-트리, XB-트리는 이러한 사항을 고려하지 않은 인덱스라고 볼 수 있다. 즉, 같은 엘리먼트에 따라 각각의 트리를 만들 때 이들 엘리먼트가 가지고 있는 구조적 관계가 완전히 사라진다는 것이다. 다시 말해서, 원본 XML 문서 안에 있는 각 엘리먼트 사이의 조상-자손, 부모-자식 관계가 사라지기 때문에 이를 이용하는 알고리즘에서 조인할 때 관계 조건을 검사하는 횟수가 늘어나게 된다. 예를 들면, a1-c1-a2-c2와 같은 간단한 XML 트리 모델이 있을 때 질의가 a//c와 같다면, 기존의 알고리즘에서 조인할 때 노드 사이의 비교가 a1-c1, a1-c2, a2-c2와 같이 일어난다. 즉 거의 모든 노드를 비교한다. 하지만 원본 문서의 구조적 관계를 최대한 유지하게 된다면, a1-c1, a2-c2만 비교하면 된다. 왜냐하면, c2는 c1의 자손이기 때문에 a1의 자손 관계를 만족시킴을 암시적으로 알 수 있다. 우리는 이러한 구조적 관계를 유지하는 데이터 모델을 구조적 인덱스(SI-트리)라고 정의하며 특징에 대해서는 3.2절에서 자세히 기술한다. 하지만, 이와 비슷하게 B^+-트리는 Containment Forest를 구성하지만, 알고리즘에서는 충분히 장점을 흡수하지 못하였다. 이들 인덱스에 대한 결점을 정리하면, 질의 노드에 대한 입력리스트를 빠르게 수집하기 위해서 역 인덱스를 이용하고 버킷의 데이터 구조는 B^+-트리 또는 XB-트리를 사용한다. 하지만 같은 엘리먼트들 간의 구조적 관계가 사라지는 결점이 있어서 알고리즘에서 조인을 효율적으로 처리할 수 없다.

3.2 SI-트리(Structured Index Tree)

앞 절에서 논의한 알고리즘에는 인덱스 쪽에 결점이 있는 것을 볼 수 있었다. 그래서 본 논문에서는 이러한 결점을 보완하기 위해서 SI-트리 인덱스를 제안한다. 다음 절에 있는 SISJ 알고리즘은 SI-트리에 기반으로 하여 수직 표현식에 대한 질의를 효율적으로 처리할 수 있게 한다.

(그림 6)은 directory-file의 XML 문서에 대한 XML 트리를 보여주고 있다. 또한 XML 문서에 대한 번호 체계를 한



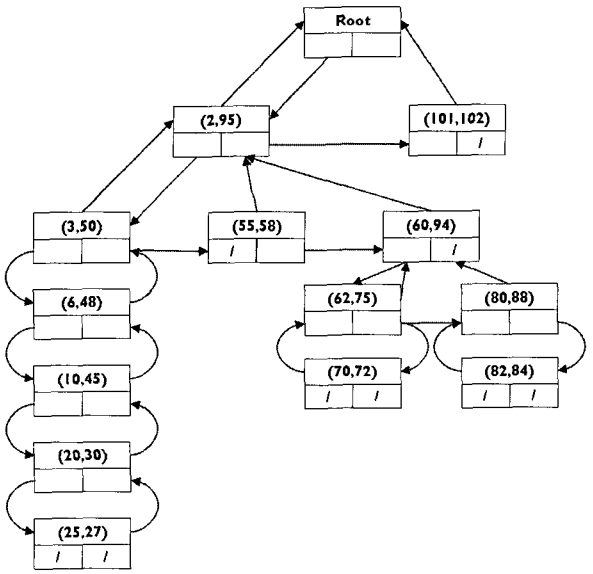
(그림 6) directory-file에 대한 XML 트리

것이다. 단, 편의상 문서번호, 레벨은 생략하였다. 본 논문에서는 하나의 XML 문서에 대해서만 고려하여 알고리즘을 구성하였다. 이것은 알고리즘의 이해를 돕기 위한 것이며 다중 문서에 대한 확장은 문서번호를 검사함으로써 쉽게 확장할 수 있다. 또한, 레벨은 부모-자식의 구조적 관계를 검사하기 위한 것으로 조상-자손의 구조적 조인 알고리즘으로부터 쉽게 확장될 수 있다.

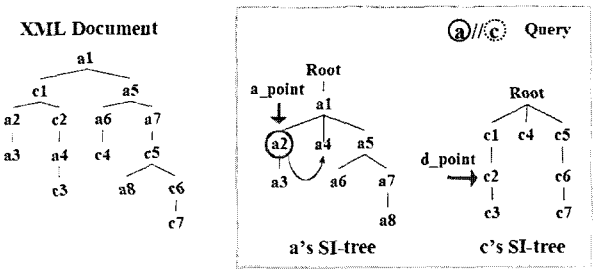
(그림 6)의 directory 엘리먼트에 대한 SI-트리 인덱스는 (그림 7)과 같이 구축된다. 트리는 최상위 노드에 가상의 루트를 가진다. 이는 한 문서 안에 같은 엘리먼트가 같은 레벨에 있을 수 있기 때문이다. 트리의 각 노드는 (key, parent[x] point, left-child[x], right-sibling[x] point)로 구성된다. 키는 노드의 시작번호와 종료번호의 쌍이며 parent[x] point는 노드 x의 부모 포인터를 지정한다. left-child[x] point는 노드 x의 첫 번째 자식노드를 지정하며 right-sibling[x] point는 노드 x의 형제노드를 지정한다. 만약 left-child[x]가 널(NULL)이 되면 노드 x에는 자식 노드가 없게 되며, right-sibling[x]가 널이 되면 노드 x에는 이어지는 형제 노드가 없음을 나타낸다.

SI-트리를 구성함으로써 같은 엘리먼트들 간의 구조적 관계를 그대로 유지할 수 있다. SI-트리는 두 가지의 장점을 가지게 된다. 첫째, 서브트리에 대한 가지치기 : 결과 값에 참여하지 않는 노드들에 대한 가지치기를 할 수 있다. 두 번째, 포함관계 룰 : 조상에 대한 SI-트리의 노드 x가 자손 노드에 대한 SI-트리의 노드 y의 조상이 되면 포함관계에 의해서 y의 서브트리에 있는 모든 노드들도 x의 자손 노드가 된다. 또한 노드 x의 조상 노드들은 y를 포함한 서브트리에 있는 모든 노드들의 조상 노드가 된다.

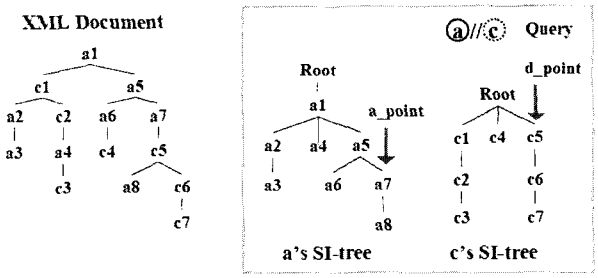
(그림 8)은 첫 번째 장점인 서브트리에 대한 가지치기가 어떻게 이루어지는지 보여주기 위한 그림이다. SI-트리는



(그림 7) directory 엘리먼트에 대한 SI-트리



(그림 8) 서브트리에 대한 가지치기



(그림 9) 포함관계 룰

그림과 같이 단순하게 표현하였다. 엘리먼트 a와 c로 이루어진 XML 문서로부터 a와 c에 대한 SI-트리가 구축되고 a//c와 같은 수직 표현식이 주어지면 a와 c사이의 구조적 관계를 결정하기 위해서 트리를 탐색하게 된다. a_point는 a의 SI-트리에 대한 포인터이며 d_point는 c의 SI-트리에 대한 포인터이다. a_point가 a2를 가리키고 있고 d_point는 c1을 가리키고 있을 경우에, c2는 문서상 a2의 다음에 오기 때문에 a2의 모든 자손들은 c2보다 앞에 오는 엘리먼트가 된다. 따라서 a2의 서브트리를 탐색할 필요가 없이 가지치기를 할 수 있다.

(그림 9)는 SI-트리 안에 있는 노드들 사이의 구조적 관계를 이용한 포함관계 룰을 보여준다. a_point가 a7을 d_point

가 c5를 가르치고 있을 때, 두 포인터가 가르치는 노드사이의 구조적 관계가 조상-자손 관계를 만족시키면 포함관계에 의해서 a7//c5, a7//c6, a7//c7, a5//c5, a5//c6, a5//c7, a1//c5, a1//c6, a1//c7 관계를 도출할 수 있다. 이는 다른 노드들에 대한 비교를 하지 않기 때문에 효율적으로 트리를 탐색할 수 있다(순차 탐색을 하지 않음). 이와 같은 장점이 많이 부각되는 XML 문서는 같은 엘리먼트들이 재귀적인 구조를 가질 경우이다.

3.3 SISJ 알고리즘(Structured Index Structural Join Algorithm)

이번 절에서 우리는 SISJ로 지칭되는 SI-트리기반의 구조적 조인 알고리즘을 제안한다. 우리는 알고리즘의 간단함과 명료함을 유지하기 위해서 단일 문서를 처리하는 XPath 표현식만을 고려한다. 알고리즘의 주요 목적은 SI-트리를 효율적으로 탐색하는 것이다. 기본적으로 SISJ 알고리즘은 깊이 우선 탐색 방법을 선택하였다. 먼저 3.1절에서 설명한 a_point와 d_point의 의미를 기억하자. 깊이 우선 탐색을 진행하기 위해서 (그림 10)에서와 같은 4가지 조건을 가지게 된다. 4가지 조건은 a_point와 d_point가 가리키는 노드 사이의 관계에 의해서 구분된다. 룰은 If/Then으로 나누어지게 되는데 If는 a_point와 d_point 사이의 관계를 나타내고 이러한 관계를 만족시킬 경우의 행동을 Then에서 기술한다. 이러한 4가지 조건에 따라서 TC1, TC2, TC3, TC4의 룰은 다음과 같다.

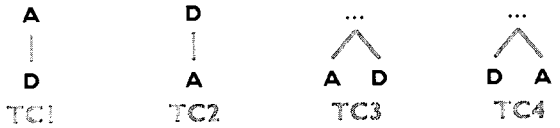
(1) TC1 : Ancestor-Descendant Relationship
- If(a_point.start < d_point.start) and (a_point.end > d_point.end)
  then, a_point.left-child != NULL → a_point = a_point.left-child
  else, a_point.left-child == NULL → a_point = selectRightSibling(a_point)

(2) TC2 : Descendant-Ancestor Relationship
- If(a_point.start > d_point.start) and (a_point.end < d_point.end)
  then, d_point.left-child != NULL → d_point = d_point.left-child
  else, d_point.left-child == NULL → d_point = selectRightSibling(d_point)
  a_point = selectRightSibling(a_point)

(3) TC3 : Sibling 1 : Former than D
- If(a_point.end < d_point.start)
  then, selectRightSibling(a_point) == NULL
  → d_point = selectRightSibling(d_point)
  else, a_point = selectRightSibling(a_point)

(4) TC4 : Sibling 2 : Latter than D
- If(a_point.start > d_point.end)
  then, selectRightSibling(d_point) == NULL
  → a_point = selectRightSibling(a_point)
  else, d_point = selectRightSibling(d_point);

SISJ에서 4개의 조건에 따라 a_point, d_point의 탐색이 결정된다. 서브트리에 대한 가지치기가 일어나는 경우는 탐색 조건 TC3와 TC4일 경우에 일어난다. TC3 조건은 조상의 SI-트리에서 서브트리에 대한 가지치기가 일어나고, TC4에서는 자손의 SI-트리에서 서브트리에 대한 가지치기가 일어나게 된다. selectRightSibling함수는 현재 노드의 모든 조상노드를 검색하여 가장 먼저 만나는 형제노드를 찾기 위한 것이다. 알고리즘은 <표 1>과 같다.

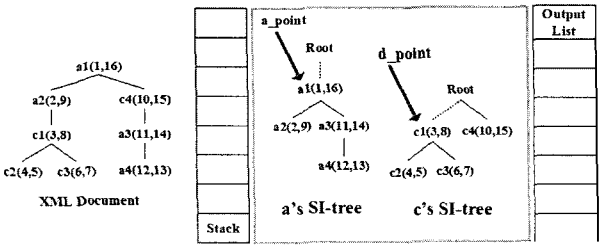


(그림 10) 4개의 SI-트리 탐색 조건

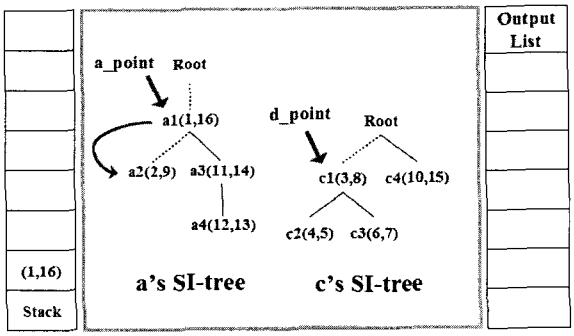
<표 1>에서 보는 것처럼 알고리즘은 함수의 인수로 조상 및 자손 SI-트리인 AT, DT가 각각 들어온다. 그리고 d_point에 대한 조상 노드들을 유지하기 위해서 하나의 조상 스택을 초기화한다. 01-02에서 a_point와 d_point를 AT, DT의 루트의 left-child로 초기화를 진행한다. 그리고 a_point 또는 d_point가 널(NULL)이 될 때까지 루프를 반복한다. 루프를 돌면서 탐색 조건 TC1 - TC4를 체크하면서 결과 값을 수집한다. 04-08은 TC1을 만족할 경우에 해당되며 a_point와 d_point사이의 관계가 조상-자손 관계이므로 이미 생성된 조상 스택에 a_point가 가리키는 노드를 삽입한다. 그리고 a_point.left-child가 NULL인지를 체크하여 a_point를 다른 노드로 이동한다. 코드 09-15는 TC2의 조건에 해당되는 코드로서 d_point.left-hcild가 NULL일 경우에 calculateOutput 함수에 의해 조상 스택과 d_point사이의 조상-자손 관계를 계산하여 output 스택에 삽입하고 a_point와 d_point를 이동한다. 그리고 코드 16-24는 TC3에 대한 코드로서 결과 쌍

<표 1> SISJ 알고리즘

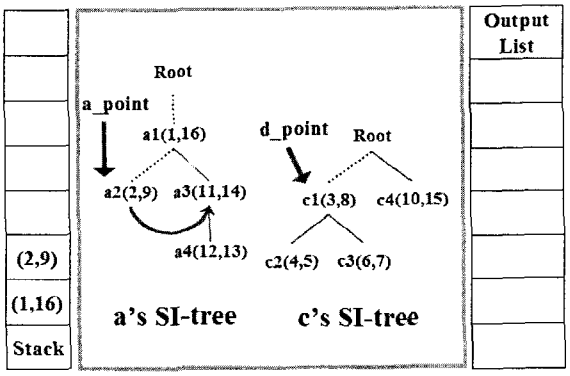
```
Input : AT & DT in Query (Vertical Expression)
   - AT : Ancestor's SI-tree
   - DT : Descendant's SI-tree
Output : All matching (q1, q2, q3, … , qn)

Algorithm SISJ(AT, DT)
01 a_point = AT.Root.left-child
02 d_point = DT.Root.left-child
03 while( a_point != NULL  &&  d_point != NULL ) {
04   if ( TC1 ) {
05     stack.insert ( a_point );
06     if ( a_point.left-child != NULL )  a_point = a_point.left-child;
07     else  a_point = selectRightSibling( AT, a_point );
08   }
09   else if ( TC2 ) {
10     if ( d_point.left-child == NULL ) {
11       output.insert ( calculateOutput ( a_point, d_point ) );
12       d_point = selectRightSibling( DT, d_point );
13       a_point = selectRightSibling( AT, a_point );
14     } else d_point = d_point.left-child;
15   }
16   else if ( TC3 ) {
17     output.insert ( calculateOutput ( a_point, d_point ) );
18     if( selectRightSibling (AT, a_point ) == NULL )
19       d_point = selectRightSibling(DT, d_point);
20     else
21       a_point = selectRightSibling (AT, a_point);
22     if( !( stack.top().start < d_point.start && stack.top().end >
        d_point.end ) )
23       stack.pop();
24   }
25   else if ( TC4 ) {
26     output.insert ( calculateOutput ( a_point, d_point ) );
27     if( selectRightSibling(DT, d_point) == NULL )
28       a_point = selectRightSibling(AT, a_point);
29     else
30       d_point = selectRightSibling (DT, d_point );
31     if( !( stack.top().start < d_point.start && stack.top().end >
        d_point.end ) )
32       stack.pop();
33   }
34 }
```



(그림 11) XML 트리와 SISJ의 초기 상태



(그림 12) SISJ의 절차 - 1

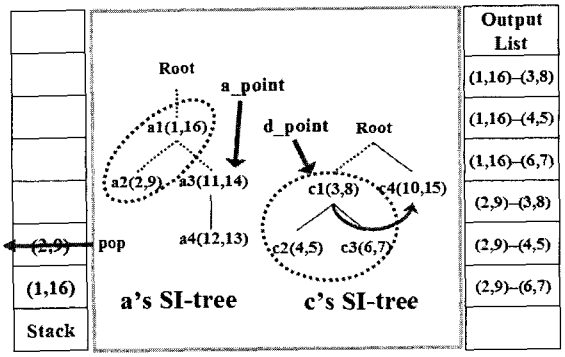


(그림 13) SISJ의 절차 - 2

을 계산한 다음에 output 스택에 저장하고 a_point와 d_point를 조건에 따라 이동한다. 마지막으로 TC4는 코드 25-33에 해당되며 TC3와 비슷하게 실행된다. SISJ의 이해를 돕기 위해서 (그림 11)부터 (그림 15)까지 도식화하였다.

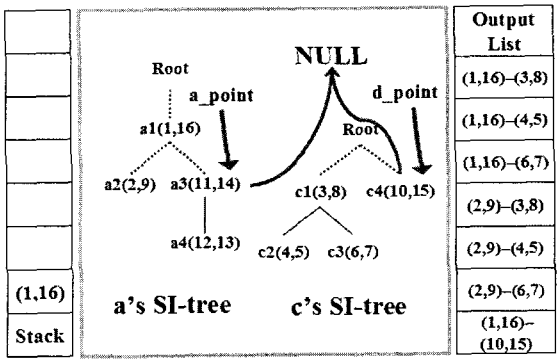
(그림 11)은 알고리즘의 초기화 상태이다. d_point의 조상 노드 리스트를 유지하기 위해서 조상 스택을 두게 된다. 이것은 포함관계를 이용한 노드 사이의 비교에서 조상 노드에 대한 이중 탐색을 방지하기 위한 것이다. 그래서 d_point의 조상이 되는 노드는 조상 스택에 삽입된다.

(그림 12)에서는 a_point와 d_point의 관계가 탐색 조건 TC1 (Ancestor-Descendant)을 만족시키기 때문에 a_point = a_point.left-child로 할당되고 a1는 c1의 조상이기 때문에 a1을 조상 스택에 삽입한다. 그리고 TC1 룰에 의해 a_point를 a2로 이동한다. (그림 13)에서는 탐색조건 TC1(Ancestor-Descendant)을 만족시키며 a_point.left-child == NULL이기 때문에 a2의 형제노드로 a_point를 이동한다. 그리고 a2를 조상 스택에 삽입한다.



Subsumption

(그림 14) SISJ의 절차 - 3



Pruning Subtrees

(그림 15) SISJ의 절차 - 4

(그림 14)는 노드 a3와 노드 c1이 탐색 조건 TC4 (Latter than D)를 만족하기 때문에 조상 스택과 d_point가 가리키는 c1사이의 결과 쌍을 계산한다. 조상 스택에 있는 노드들과 c1을 포함한 서브트리의 노드들 사이에는 조상-자손 관계에 있기 때문에 이에 대한 결과 값을 출력 목록(Output List)에 삽입한다. 그리고 d_point를 selectRightSibling함수에 의해 c4노드로 이동한다. 이때 포함관계 룰(그림 9)에 의해서 c1의 서브트리를 결과 쌍에 반영할 뿐 탐색을 더 이상 하지 않는 가지치기 효과를 가질 수 있게 된다.

(그림 15)는 a_point와 d_point의 노드사이에 TC2 (Descendant-Ancestor)관계가 성립된다. d_point.left-child == NULL 이기 때문에 a_point = selectRightSibling(a_point), d_point = selectRightSibling(d_point)에 의해서 a_point=d_point=NULL이 된다. 그리고 a_point와 d_point가 널이기 때문에 알고리즘은 종료하게 된다. 단, 종료하기 전에 조상 스택이 비워있지 않으면 d_point가 가리키는 c4사이에 조상-자손 관계가 아직 존재하기 때문에 결과 값을 계산하고 출력 목록에 삽입한다.

4. 실험 평가

SISJ 알고리즘의 성능 평가를 위해서 XB-트리 기반의 TwigStackXB 알고리즘과 비교분석을 한다. 기존 논문[18]

에서 수직 표현식에 대해서 TwigStackXB 알고리즘이 B^+-트리 기반 구조적 알고리즘보다 성능이 높다고 평가되었기 때문에 본 논문에서는 TwigStackXB 알고리즘을 비교대상으로 한다.

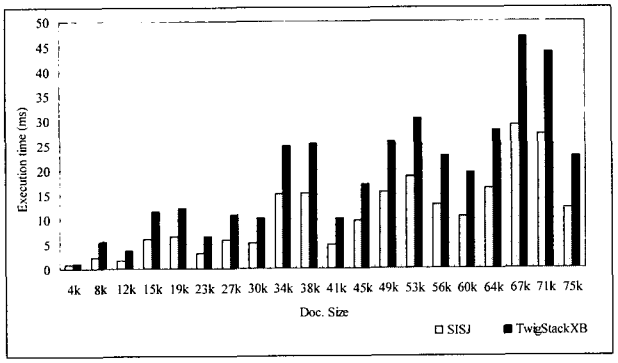
실험 환경 설정을 위해서 IBM에서 개발한 ToXgene를 이용하여 임의의 XML 문서를 생성한다. 알고리즘 자체가 수직 관계(조상-자손)만을 질의 대상으로 하기 때문에 Person, Children 엘리먼트만을 가지고 XML 문서를 랜덤하게 생성시킨다. 생성되는 엘리먼트의 개수는 200개에서 4000개 사이로 하며 이에 대한 문서의 크기는 4k부터 75k에 해당된다. Person과 Children 엘리먼트 사이의 재귀성에 따라서 문서를 3종류로 분류한다. 만약 트리 높이가 3이면 재귀 관계는 Person-Children-Person, Person-Person-Children, Person-Children-Children와 같은 구조를 이루게 된다. 첫 번째는 Person과 Children의 재귀적 관계가 높을 경우 (자식 노드의 수 : 50, 트리 높이 : 100), 두 번째는 재귀적 관계가 비교적 적을 경우 (자식 노드의 수 : 50, 트리 높이 : 3), 세 번째는 재귀적 관계가 전혀 없을 경우 (트리 높이 : 2)로 분류한다. 세 종류의 문서에 대한 실험을 통해서 수직 표현식을 처리하기 위한 알고리즘의 성능을 비교분석할 수 있다.

Person과 Children사이의 재귀적 관계가 높은 경우의 실험은 (그림 16)과 같다. 트리의 높이가 100일지라도 Person-Children사이의 재귀적 관계가 문서의 크기에 비례해서 높아지는 것이 아니기 때문에 그림 16과 같은 성능을 보여준다. 예를 들면, a1-a2-a3-a4-c1-c2-c3-c4, a1-c1-a2-c2-a3-c3-a4-c4와 같은 경우에 같은 높이 일지라도 전자는 재귀적 관계가 없는 경우이고 후자는 재귀적 관계가 높은 경우에 해당된다. 따라서 SISJ와 TwigStackXB의 성능이 불규칙함을 띠고 있음을 그래프에서 보여준다. 그러나 대체적으로 문서의 크기에 따라 수행시간도 증가함을 볼 수 있다. TwigStackXB보다 30-40%의 성능향상이 있는 것을 볼 수 있는데, 이유는 SI-트리를 구축함으로써 조상 SI-트리와 자손 SI-트리의 가지치기가 효율적으로 처리되며, 또한 포함관계 룰을 통한 조상-자손 관계의 비교 횟수를 효과적으로 줄이기 때문으로 분석된다.

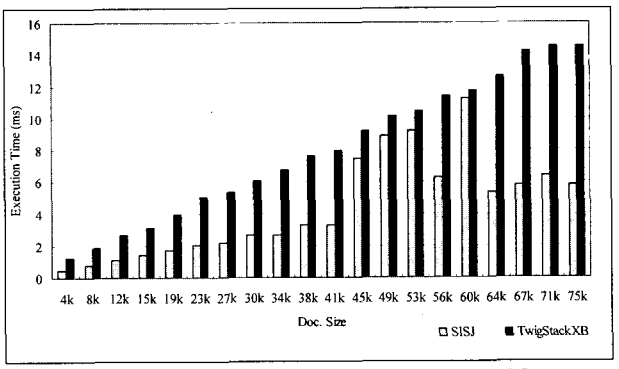
(그림 17)은 엘리먼트간 재귀적 관계가 낮은 경우의 실험이다. 재귀적 관계가 낮은 경우에도 SISJ가 TwigStackXB보다 성능이 우수함을 볼 수 있다. 전체적으로 문서의 크기가 증가함에 따라 TwigStackXB는 실행시간이 일정하게 증가됨을 볼 수 있다. 하지만 SISJ는 엘리먼트간 재귀 관계에 민감하게 성능이 좌우됨을 볼 수 있다. 문서 45k, 49k, 53k, 60k에서 TwigStackXB와 비슷한 성능을 내는 이유는 생성된 문서에서 엘리먼트 간 재귀적 관계가 적은 경우이기 때문이다.

Person과 Children사이의 재귀적 관계가 없는 경우의 실험은 (그림 18)과 같다. 주의 깊게 볼 점은 재귀적 관계가 없는 문서에서도 SISJ가 우수한 성능을 냄을 볼 수 있다. 이에 대한 분석을 위해서 (그림 16-18)을 SISJ의 실행시간을 TwigStackXB의 실행시간으로 나누어서 상대비교 해보기로 한다.

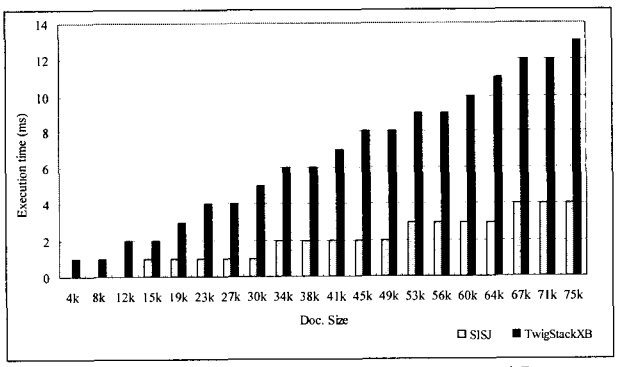
Person-Children 엘리먼트 문서에 재귀성이 없는 경우에는 SI-트리와 XB-트리가 알고리즘의 성능에 거의 영향을 주지



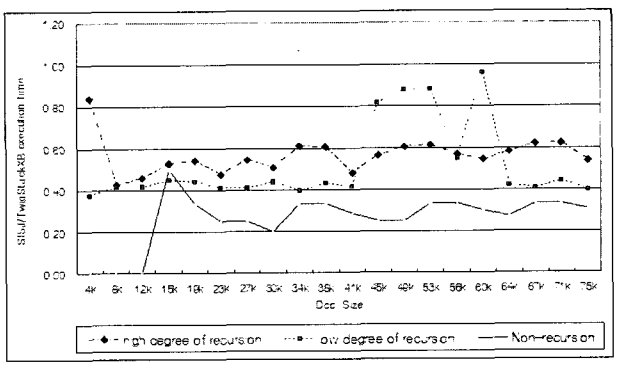
(그림 16) 엘리먼트간 재귀적 관계가 높은 경우



(그림 17) 엘리먼트간 재귀적 관계가 낮은 경우



(그림 18) 엘리먼트간 재귀적 관계가 없는 경우



(그림 19) SISJ/TwigStackXB 실행 타임에 대한 상대비교

않는다. 왜냐하면 재귀성이 없기 때문에 가지치기가 일어나는 부분이 없어서 순차검색을 하기 때문이다. (그림 19)와 같이 재귀성이 없는 문서에서 SISJ의 성능이 TwigStackXB 보다 성능이 더 우수한 것은 알고리즘의 복잡도 때문이다. 다시 말해서, SISJ는 루프를 돌면서 조건만 검사하여 결과를 수집하지만, TwigStackXB는 조상-자손 관계를 계산하기 위해 2개의 스택을 유지하는 비용과 재귀함수를 이용하여 조상-자손 관계를 검사하기 때문이다. 그리고 (그림 16, 17, 18)에서 보는 것처럼 상대적으로 재귀적 관계가 높은 문서 일수록 실행시간이 더 걸림을 알 수 있다.

5. 결론 및 향후 과제

XML은 데이터베이스와 같은 완전 구조화된 데이터뿐만 아니라 웹 문서 같은 반 구조화된 데이터를 표현하는데 널리 사용되고 있기 때문에 많은 애플리케이션 영역에서 효율적으로 이용되고 있는 상황이다. 예를 들어, 데이터 저장소, 데이터 표현, 전송 메시지 등에 이용된다. XPath 표현식이 처리되기 위해서는 XML 문서의 트리 모델 안에 있는 모든 후보 노드들이 조건을 만족하는지를 검사해야 된다. 결과적으로, 효율적인 XPath 질의 처리를 위해서는 트리 모델에 대한 탐색 비용을 최소화하는 것이 중요하다. 이러한 관점에서 우리는 원본 XML 트리 모델 내에서 되도록 많은 구조적 관계를 유지하도록 하는 구조적 인덱스(SI)를 제안하고 이를 기반으로 하는 새로운 구조적 조인 알고리즘(SISJ)을 제안하였다. 추후 연구로서, 우리는 복잡한 표현식 질의를 수직 표현식으로 분해하는 것 없이 효율적으로 처리할 수 있는 기법에 대한 연구를 계획하고 있다.

참 고 문 헌

[1] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, Eve maler, "Extensible Markup Language (XML) 1.0 (Second Edition), W3C Recommendation 6 October 2000." See http://www.w3.org/TR/REC-xml/.

[2] Howard Katz, "XQuery From The Experts - a guide to the W3C XML Query Language," Addison-Wesley, 2003.

[3] Mary Fernandez, Ashok Malhotra, et al., "XQuery 1.0 and XPath 2.0 Data Model(XDM)," W3C Candidate Recommendation 3 November 2005. See http://www.w3.org/TR/xpath-datamodel/.

[4] Anders berglund, Scott Boag, et al., "XML Path Language (XPath) 2.0, W3c Candidate Recommendation 3 November 2005. See http://www.w3.org/TR/xpath20/.

[5] Chun Zhang, et al., "On Supporting Containment Queries in Relational Database Management Systems", ACM SIGMOD, May 2001.

[6] Al-Khalifa, S., Jagadish, H.V., Koudas, N., Patel, J.M., Srivastava, D., Yuqing Wu, "Structural Joins : A Primitive for Efficient XML Query Pattern Matching," pp 141-152, 2002, ICDE 2002.

[7] Shu-Yao Chien, Zografoula Vagena, Donghui Zhang, "Efficient Structural Joins on Indexed XML Documents," VLDB, 2002.

[8] Nicolas Bruno, Nick koudas, Divesh Srivastava, "Holistic Twig Joins: Optimal XML Pattern Matching," ACM SIGMOD, June 2002.

[9] Antonm Guttman, "R-Trees: A Dynamic Index Structure For Spatial Searching," ACM SIGMOD, 1984.

[10] Hanyu Li, Mong Li Lee, Wynne Hsu, Chao Chen, "An Evaluation of XML Indexes for Structural Join," SIGMOD Record, Vol.33, No.3, September 2004.

[11] Quanzhong Li, Bongki Moon, "Indexing and Querying XML Data for Regular Path Expressions," pp 361-370, VLDB 2001.

[12] Sudipto Guha, H.V. Jagadish, et al., "Approximate XML Joins," SIGMOD 2002.

[13] S. Ceri, P. Fraternali, S. Paraboschi, "XML: Current Developments and Future Challenges for the Database Community," EDBT 2000.

[14] Tova Milo, Dan Suciu, "Index Structures for Path Expressions," pp.277-295, ICDT 1999.

[15] G. Graefe, "Query evaluation techniques for large databases," ACM Computing Surveys, 25(2), 1993.

[16] J. McHugh and J. Widom, "Query optimization for XML," In Proceedings of VLDB, 1999.

[17] J. Shanmugasundaram, E. J. Shekita, R. Barr, M. J. Carey, B. G. Lindsay, H. Pirahesh, and B. Reinwald, "Efficiently publishing relational data as XML documents," In Proceddings of VLDB, 2000.

[18] E. Shekita and M. Carey, "A performance evaluation of pointer based joins," Proceedings of SIGMOD, 1990.

[19] Denilson Barbosa, Alberto Mendelzon, etc., "ToXgene - the ToX XML Data Generator," IBM, See http://www.cs.toronto.edu/tox/toxgene/.

김 학 수

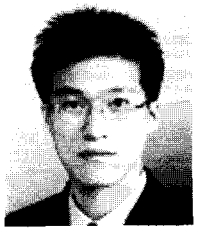
e-mail : hagsoo@cse.hanyang.ac.kr

2004년 한양대학교 전자 컴퓨터 공학과 (학사)

2006년 한양대학교 컴퓨터공학과 (석사)

2006년~현 재 한양대학교 컴퓨터공학과 박사과정

관심분야 : 데이터베이스, 시멘틱 마이닝, e-비즈니스



신 영 재

e-mail : yjshin@cse.hanyang.ac.kr

2006년 한양대학교 전자컴퓨터공학부 (학사)

2006년~현 재 한양대학교 컴퓨터공학과 석사과정

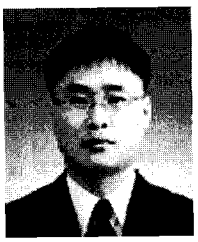
관심분야 : Mobile, 데이터베이스, 데이터 마이닝

황 진 호

e-mail : jhhwang@database.hanyang.ac.kr
2006년 상지대학교 컴퓨터공학부(학사)
2006년~현 재 한양대학교 컴퓨터공학과 석사과정
관심분야 : Mobile, 데이터베이스, 파일 시스템

이 승 미

e-mail : smlee@database.hanyang.ac.kr
1989년 숭실대학교 전산학과(학사)
1991년 한국과학기술원 전산학과(석사)
1999년 한국과학기술원 전산학과(박사)
1991년 1월~1993년 12월 충남대학교 정보통신공학과 시간강사
2000년 3월~2001년 3월 엘엔에치이치코리아(주) 음성언어기술연구소 책임연구원
2006년 3월~현 재 한양대학교 컴퓨터공학과 BK21 사업팀 연구원



손 진 현

e-mail : jhson@hanyang.ac.kr
1996년 서강대학교 전산학과(학사)
1998년 한국과학기술원 전산학과(석사)
2001년 한국과학기술원 전자전산학과(박사)
2001년 9월~2002년 8월 한국과학기술원 전자전산학과 박사후 연구원
2002년 9월~현 재 한양대학교 컴퓨터공학과 조교수
관심분야 : 데이터베이스, e-비즈니스, 유비쿼터스 컴퓨팅, 임베디드 시스템