빠른 XML 질의 처리를 위한 세그먼트 조인 기법
(Segment Join Technique for Processing XML Queries Fast)

배 진 욱 † 문 봉 기 †† 이 석 호 †††
(Jinuk Bae) (Bongki Moon) (Sukho Lee)

요 약 XML 데이타를 대상으로 선형 질의나 가지모양 질의 같은 복잡한 질의가 많이 연구되고 있다. 이와 같은 질의를 처리하기 위해 XML 데이타를 구조정보에 의해 미리 인코딩한 후, 질의 처리시 구조정보를 이용하여 빠르게 질의를 수행하는 구조 조인 알고리즘들이 제안되었다. 그 중 최근에 제안된 TwigStack 알고리즘과 TSGeneric⁻ 알고리즘은 각각 인덱스가 없는 환경과 있는 환경에서 수행시간이 입력 데이타의 양과 비례하는 최적의 성능을 보여주었다. 하지만 이들 알고리즘은 질의의 길이(질의에 나타난 엘리먼트 개수)에 비례하여 입력 데이타의 양이 증가하고, 따라서 수행시간이 길어진다는 제한점이 있다.

이 논문에서는 기존의 구조 조인 알고리즘들에 구조 인덱스를 결합한 세그먼트 조인 기법을 제안한다. 이 기법은 질의 노드와 노드 간의 구조 조인과는 달리, 구조 인덱스를 이용하여 일련의 질의 노드들을 하나의 세그먼트로 식별한 후 세그먼트와 세그먼트 사이의 조인을 수행한다. 그 결과 세그먼트마다 하나의 질의 노드만을 읽음에 의해 질의를 처리할 수 있게 되어 수행성능이 향상된다. 다양한 데이타셋에 대해 인덱스가 없는 환경에서 실험 결과, 세그먼트 조인 기법을 적용한 SegmentTwig 알고리즘은 TwigStack 알고리즘보다 우수한 성능을 보였다.

키워드 : XML 데이타베이스, 가지모양 질의, 세그먼트 조인, 세그먼트트위그 알고리즘

Abstract Complex queries such as path and twig patterns have been the focus of much research on processing XML data. Structural join algorithms use a form of encoded structural information for elements in an XML document to facilitate join processing. Recently, structural join algorithms such as TwigStack and TSGeneric⁻ have been developed to process such complex queries, and they have been shown that the processing costs of the algorithms are linearly proportional to the sum of input data. However, the algorithms have a shortcoming that their processing costs increase with the length of a query. To overcome the shortcoming, we propose the *segment join technique* to augment the structural join with structural indexes such as the 1-Index. The *SegmentTwig* algorithm based on the segment join technique performs joins between a pair of segments, which is a series of query nodes, rather than joins between a pair of query nodes. Consequently, the query can be processed by reading only a query node per segment. Our experimental study shows that segment join algorithms outperform the structural join methods consistently and considerably for various data sets.

Key words : XML Database, Twig Query, Segment Join, SegmentTwig Algorithm

1. 서 론

XML 질의 처리 방법은 구조 인덱스(structural index)를 이용하는 방법과 넘버링 스킴(numbering scheme)을 사용하는 방법으로 나눌 수 있다. 구조 인덱스 기법은 XML 문서의 구조 정보를 미리 요약해놓은 뒤 이를 통해 빠른 질의 처리를 하려는 시도이다. 이 기법의 예로는 DataGuide[1], 1-Index[2], APEX[3], F+B 인덱스[4], F&B 인덱스[5]가 있다.

넘버링 스킴 기법은 XML 데이타의 각 노드마다 위치 정보에 의해 한 쌍의 값을 부여하고 이 값을 이용하여 질의 처리를 한다. 이 때, 동일한 이름을 갖는 노드들에 부여된 값들은 하나의 테이블로 저장된다. 즉, XML 문서에 여러 author 노드들과 여러 title 노드들이

· 본 연구는 2005년도 두뇌한국21사업과, 정보통신부의 대학 IT연구센터(ITRC) 지원을 받아 수행되었습니다.
† 학생회원 : 서울대학교 전기컴퓨터공학부
· jinuk@db.snu.ac.kr
†† 비 회 원 : University of Arizona Department of Computer Science Associate Professor
bkmoon@cs.arizona.edu
††† 종신회원 : 서울대학교 컴퓨터공학부 교수
shlee@cse.snu.ac.kr
논문접수 : 2004년 10월 18일
심사완료 : 2005년 3월 5일

나타날 수 있는데, 각 노드들에 부여된 값들은 author 테이블과 title 테이블로 나뉘어 저장된다. 넘버링 스킴의 종류로는 pre-and-postorder 방법[6], 확장 preorder 방법[7], 그리고 여는 태그(opening tag)와 닫는 태그(closing tag)의 위치 정보에 의한 방법[8] 등이 있으며, 넘버링 스킴에 의해 부여된 값을 기반으로 구조 조인(structural join)을 통해 질의를 처리하는 다양한 알고리즘들이 제안되었다[7-14]. 이 중 선형 질의(path query)를 위한 PathStack 알고리즘과 가지모양 질의(twig query)를 위한 TwigStack 알고리즘은 인덱스가 없는 테이블에 대해 최적이라고 알려져 있으며[10], TSGeneric⁻ 알고리즘은 XR 트리 인덱스를 통해 입력 데이타의 양을 줄임에 의해 성능을 개선하였다[12,13]. 하지만, 이들 구조 조인 알고리즘은 질의의 길이(질의에 나타난 엘리먼트의 개수)에 비례하여 입력 데이타의 양이 증가하고, 그 결과 수행시간이 증가한다는 제한점이 있다. 예를 들어, XPath 문법[15]에 따라 작성된 다음의 선형 질의와 가지모양 질의를 살펴보자.

- q_{path} = /site/regions//items
- q_{twig} = /site/regions/namerica/item[@id = "item20748"]/name

q_{path}와 q_{twig}는 XMark 벤치마크[16]에 포함된 질의다. 이 벤치마크에서는 총 20개의 질의가 사용되는데, q_{path}는 두 번째로 짧은 질의이며, 가장 긴 질의는 길이가 12이다. q_{path}를 처리하기 위해 PathStack 알고리즘은 site와 regions와 items라는 3개의 테이블을 접근해야 하고, TwigStack과 TSGeneric⁻ 알고리즘은 q_{twig}를 처리하기 위해 site, regions, namerica, item, id, name이라는 6개의 테이블과 "item20748"을 저장하고 있는 텍스트 테이블을 접근해야 한다. 다만, TwigStack 알고리즘은 테이블 내의 모든 데이타를 읽어야 하지만, TSGeneric⁻ 알고리즘은 XR 트리를 사용함으로써 테이블 내의 일부 데이타를 읽지 않을 수 있다.

이와 같은 구조 조인 알고리즘은 질의가 길어진다면 필연적으로 더 많은 테이블을 접근해야만 하며, 따라서 수행시간이 증가하는 문제점이 존재한다. 이 문제를 해결하기 위해 이 논문에서는 구조 인덱스 기법을 넘버링 스킴 기법에 적용한다. 구조 조인에서의 노드와 노드 간의 조인이 아니라, 구조 인덱스를 사용하여 일련의 노드들을 하나의 세그먼트로 식별한 후 세그먼트와 세그먼트 사이의 조인을 수행한다. 그 결과, 선형 질의의 경우 항상 하나의 테이블만을, 가지모양 질의에는 일부의 테이블만을 (테이블 내에서 일부의 데이타가 아니라) 읽고도 질의를 처리할 수 있다. 예를 들어 q_{path}는 items 테이블만을 접근하고, q_{twig}는 item, name 테이블과 텍스트 테이블만을 접근한다.

논문의 구성은 다음과 같다. 2장에서 본 연구의 배경이 되는 넘버링 스킴과 구조 조인 알고리즘, 그리고 구조 인덱스를 살펴본다. 3장에서는 이를 바탕으로 새로운 데이타 모델을 제시하고, 이 모델에 기반한 세그먼트 조인 기법을 제안한다. 4장에서는 세그먼트 조인 기법에 기반한 SegmentTwig 알고리즘을 제시하며, 5장에서 실험을 통해 성능의 우수함을 보인다. 마지막으로 6장에서 결론을 맺는다.

2. 배 경

2.1 넘버링 스킴

XML은 데이타를 표현하는데 있어 트리 구조를 기반으로 하고 있으므로, 이 논문에서는 각 XML 문서를 XML 데이타 트리라고 부른다. XML 데이타 트리는 엘리먼트 노드와 텍스트 노드라는 두 종류의 노드로 구성된다. 엘리먼트 노드는 엘리먼트(태그)와 애트리뷰트 모두를 지칭하며, 구분할 필요가 있다면 애트리뷰트 앞에 특수문자 @를 덧붙인다. 텍스트 노드는 여는 태그와 닫는 태그 사이에 오는 문자열이나, 애트리뷰트 다음에 오는 문자열을 나타낸다.

넘버링 스킴은 XML 데이타 트리에 존재하는 노드마다 한 쌍의 값을 부여한다. 다음과 같은 간단한 XML 문서를 살펴보자.

<a>AB

이 문서는 루트 노드 a가 두 개의 b 노드를 자식으로 가지고 있고, 각 b 노드는 A와 B라는 텍스트 노드를 자식으로 가지고 있는 구조이다. 이 때, 넘버링 스킴에 의해 <a>에 1, 에 2, A에 3과 같이 오름차순에 의한 숫자를 부여할 수 있다. 그 결과 a 노드는 1을 받은 <a>와 8을 받은 에 의해 (1:8)이라는 값이 부여되고, 동일한 원리로 첫 번째 b 노드는 (2:4)를, 두 번째 b 노드는 (5:7)이 부여된다. 이 논문에서는 한 노드에 부여된 한 쌍의 숫자를 (begin:end)라고 나타내는데, (begin:end)는 구간의 의미를 가지고 있다. 이 때, 노드 n1이 노드 n2의 조상이라면 두 노드의 구간 사이에는 다음 관계가 성립한다.

(1) n_1.begin < n_2.begin & n_1.end > n_2.end

예를 들어, a 노드는 두 b 노드의 조상으로서 a의 구간 (1:8)은 두 b 노드의 구간 (2:4)와 (5:7)을 각각 포함한다. 그리고, 각 노드에 level (높이) 정보를 더 준다면 부모-자식 관계도 알 수 있다.

(2) n_1.level + 1 = n_2.level

넘버링 스킴에 의해 텍스트 노드는 하나의 값만을 부여받는데, 이 값은 begin과 end 동일한 구간을 뜻한다.

즉, 위의 예에서 텍스트 노드 A는 (3:3)이며, B는 (6:6)이다.

그림 1은 이 논문에서 예제로 사용할 XML 데이타 트리에 대해 넘버링 스킴에 의해 값이 부여된 모습을 보여준다.

2.2 구조 조인 알고리즘

구조 조인 알고리즘이 수행되기 위해서는 XML 데이타의 노드마다 넘버링 스킴에 의해 한 쌍의 값을 부여한 후, 동일한 이름을 갖는 노드들이 하나의 테이블에 저장되어야 한다. 예를 들어, XML 문서에 여러 author 노드들과 여러 title 노드들이 나타날 수 있는데, 이 노드들은 author 테이블과 title 테이블로 나뉘어 저장된다. 각 테이블은 노드의 인코딩 정보인 begin, end, level 애트리뷰트로 구성되고, 각 테이블에 저장된 노드들은 begin 값에 의해 오름차순으로 정렬되어 있어야 한다.

가지 모양 질의를 수행하는 구조 조인 알고리즘으로는 각 테이블에 인덱스가 없을 경우 최적이라고 알려진 TwigStack 알고리즘[10]과, TwigStack 알고리즘을 기반으로 테이블마다 XR 트리 인덱싱에 의해 성능을 향상시킨 TSGeneric⁻ 알고리즘[13]이 가장 빠르다고 알려져 있다. TwigStack 알고리즘은 두 단계로 구성된다. 첫 번째 단계에서는 getNext 함수에 의해 반환되는 노드들에 대해 가지 모양 질의의 한 선형 경로를 만족하는지를 검사하여, 선형 경로를 만족하는 노드쌍들을 출력한다. 두 번째 단계에서는 첫 번째 단계의 결과를 병합(merge)하여 최종 가지모양 질의의 답을 구한다. TSGeneric⁻ 알고리즘은 TwigStack 알고리즘의 getNext 함수를 XR 트리를 이용하도록 수정하였고 나머지는 동일하다.

2.3 구조 인덱스

구조 인덱스는 XML 문서 탐색을 위해 구조 정보를 요약해 놓는다. 다양한 구조 인덱스 중 1-Index[2]는 XML 데이타 트리에 존재하는 엘리먼트 경로들을 중복없이 저장한 트리이다. 그림 2는 그림 1의 데이타 트리에 대해 만들어진 1-Index로서, 데이타 트리에는 두 개의 /a/b/c 경로가 존재하지만, 1-Index에서는 이 경로들이 하나로 나타난다. 이와 같이 경로가 합쳐지므로 1-Index는 항상 데이타 트리보다 작거나 같다.

또한, 1-Index의 각 노드는 유일한 노드 식별자 nid를 갖는데, 이 논문에서는 항상 자식 노드는 부모보다 큰 nid를 부여한다. 그림 2에서 루트 노드의 a.1은 a 노드의 nid가 1이라는 의미이고, 루트에서 각 단말 노드에 이르는 경로 상에서 각 노드의 nid는 증가한다.

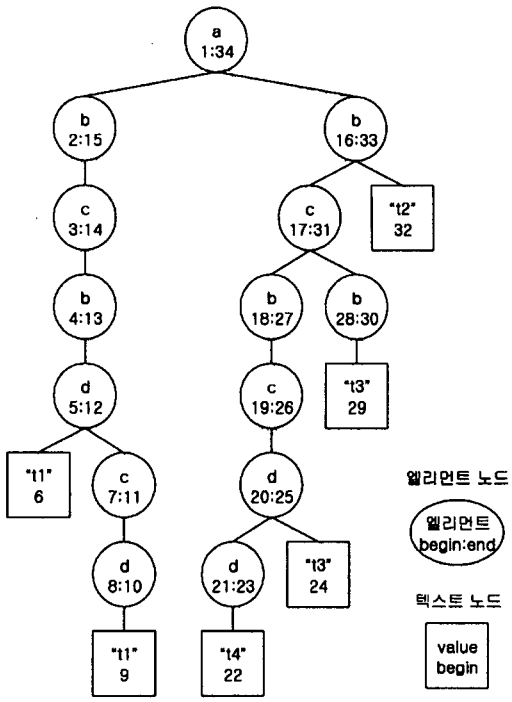

그림 1 넘버링 스킴이 적용된 XML 데이타 트리

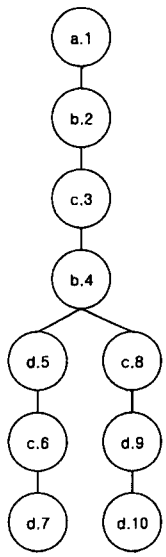

그림 2 1-Index

3. 세그먼트 조인

3.1 데이타 모델

기존의 구조 조인 알고리즘들은 XML 데이타 트리에 대해 begin, end, level을 부여한 데이타 모델을 사용하

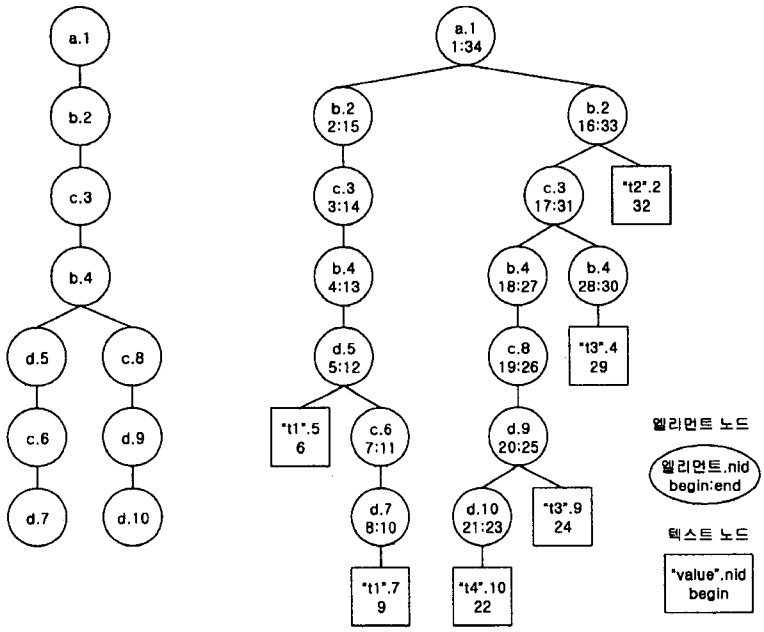

그림 3 1-Index와 1-Index에 의해 확장된 데이타 모델

였다. 이 논문에서는 이 세 값 뿐만 아니라 1-Index의 nid를 추가로 부여한 데이타 모델을 제시한다. 우리의 데이타 모델에서 엘리먼트 노드는 <element, nid, begin, end, level>로 표현된다. 노드에 부여되는 nid는 루트부터 그 노드에 이르는 경로에 의해 결정된다. 그림 3에서 2:15를 가지고 있는 b 엘리먼트 노드는 nid 2가 주어졌는데, 이는 루트에서 b 노드에 이르는 경로 /a/b에 해당하는 1-Index의 노드의 nid가 2이기 때문이다. 데이타 트리에는 /a/b를 따르는 노드가 두 개 있으며, 그래서 두 노드 모두 nid로 2를 갖는다.

한편, 텍스트 노드는 <value, begin, nid, level>로 나타낸다. 1-Index에는 텍스트 노드가 존재하지 않으므로, 동일한 경로를 가지는 노드가 존재하지 않는다. 대신 텍스트 노드는 그 부모의 경로가 질의 조건으로 사용되므로, 부모 노드의 nid를 물려받는다. 예를 들어, begin 값이 6인 "t1" 텍스트 노드는 그 부모의 nid 값인 5를 nid로 가진다.

3.2 질의의 세그먼트 분할

정의 1. 질의는 질의 세그먼트라고 명명된, 서로 겹치지 않는 선형의 부분 패턴들로 분할된다. 각 질의 세그먼트 s는 $e_1\theta_1e_2\theta_2...\theta_{n-1}e_n$으로 표시되는데, 이 때 e_i와 θ_i는 다음 조건을 만족한다.

(i) e_n은 조인트 노드거나 단말 노드거나 타겟 노드이다.

(ii) e_i $(1 \le i < n)$는 조인트 노드, 단말 노드, 타겟 노드가 될 수 없다.

(iii) θ_i $(1 \le i \le n-1)$는 조상-자손 관계(//)나 부모-자식 관계(/)를 나타낸다.

예를 들어, 질의 a/b[c//d/e]f는 그림 4와 같이 세 개의 질의 세그먼트 a/b와 c//d/e와 f로 분할된다. 이 때, b는 두 개 이상의 자식을 갖는 조인트 노드이며, e는 단말 노드, 그리고 f는 단말 노드이자 타겟 노드(XPath 문법에 의해)이다.

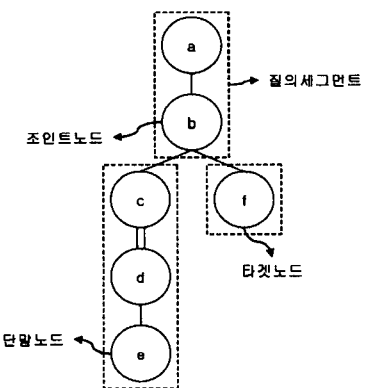

그림 4 a/b[c//d/e]/f의 세그먼트 분할

각 질의 세그먼트는 선형 질의라고 생각할 수 있는데, 1-Index는 선형 질의를 처리하는데 유용하다. 1-Index와 선형 질의와의 관계는 다음과 같다. 먼저, 1-Index에

서 주어진 선형 질의를 만족하는 모든 경로를 찾는다. 그런 후, 그 경로의 마지막 노드가 가지는 nid와 동일한 nid를 가지는 데이타 노드들을 찾으면 이 데이타 노드들은 선형 질의를 만족한다. 1-Index에서의 경로와 데이타 노드에서의 경로가 동일하기 때문이다. 예를 들어, 그림 3에서 패턴 c/d를 만족하는 경로는 1-Index에 두 개 존재한다. c.6/d.7과 c.8/d.9로서 마지막 노드의 nid로 7과 9를 얻는다. 데이타 트리에서 7 또는 9의 nid를 갖는 엘리먼트 노드(전부 d 노드임)는 패턴 c/d를 만족한다. 이 때, 7 또는 9를 nid로 하는 텍스트 노드의 경우 c/d/text() (c/d를 부모로 하는 모든 텍스트를 출력하라는 질의)를 만족한다.

3.3 세그먼트 조인

선형 질의의 경우 1-Index에서 타겟 노드의 nid를 찾으면, 이 nid 만으로 답이 되는 데이타 노드들을 얻을 수 있지만, 가지모양 질의의 경우 그렇지 않다. 예를 들어, 질의 a//b[c/d] (a를 조상으로 가지며 c/d를 자식으로 갖는 b 데이타 노드를 찾으라는 질의)를 생각해 보자. 1-Index에서 질의를 만족하는 nid는 유일하게 4가 있는데, 실제로 nid가 4인 세 개의 데이타 노드 중 오직 한 노드 <b, 4, 18, 27>만이 답이 된다. 이런 일이 발생하는 이유는 1-Index를 생성할 때 엘리먼트 경로들이 합쳐졌기 때문이다. 즉, 가지모양 질의의 경우 타겟 노드의 nid만을 저장해서는 답을 구할 수 없으므로 세그먼트 조인이 필요하다.

세그먼트 조인이란 두 질의 세그먼트 s_1과 s_2가 주어졌을 때, s_1과 s_2 사이의 조상-자손 관계 (또는 부모-자식 관계)를 만족하는 s_1의 마지막 데이타 노드와 s_2의 마지막 데이타 노드 쌍을 찾는 구조 조인으로 $s_1 \triangleright s_2$ (또는 s1▷s2)로 표기한다.

정의 2. 1-Index에서 세그먼트 조인 s1▷s2 (또는 $s_1 \triangleright s_2$)를 만족하는 s_1의 마지막 1-Index 노드와 s_2의 마지막 1-Index 노드 쌍을 조인쌍이라 정의하고, 조인쌍들의 집합을 $Join_{pair}(s_1 \triangleright s_2)$ (또는 $Join_{pair}(s_1 \triangleright s_2)$)라고 표시한다.

예를 들어, s_1=a//b와 s_2=c/d에 대해 $Join_{pair}(s_1 \triangleright s_2)$는 {(b.2,d.7),(b.2,d.9),(b.4,d.7),(b.4,d.9)}이다. 이 중 1-Index에서 (b.2,d.7)을 살펴보면, (i) b.2는 a//b를 만족하는 a.1/b.2 경로에 있으며, (ii) d.7은 c.6/d.7 경로에 있다. 또한 (ii) a.1/b.2는 c.6/d.7의 조상이므로 조인쌍에 해당한다.

정리 1. 데이타 노드쌍 (n_1, n_2)이 다음 조건 (3)과 (4)를 만족하면 $s_1 \triangleright s_2$ (또는 $s_1 \triangleright s_2$)의 답이 되고, 역도 성립한다.

(3) (n_1.nid, n_2.nid) ∈ $Join_{pair}(s_1 \triangleright s_2)$ (또는 $Join_{pair}(s_1 \triangleright s_2)$)

(4) n_1.begin < n_2.begin & n_1.end > n_2.end

조건 (3)은 노드 n_1이 s_1을 만족하며, n_2는 $s_1//s_2$ (또는 s_1/s_2)를 만족한다는 것을 의미한다. 그리고, 조건 (4)에 의해 두 노드는 동일 경로 위에 있으므로 $s_1 \triangleright s_2$ (또는 $s_1 \triangleright s_2$)를 만족한다. 조건 (4)는 구조 조인에서 사용된 조건 (1)과 동일하다는 점에서 세그먼트 조인은 구조 조인 위에 조건 (3)을 더 부가한 조인이다.

예제 1. (a//b)▷(c/d)가 주어졌을 때,

① 두 데이타 노드 <b, 4, 4, 13>과 <d, 5, 5, 12> (b.4, d.5)가 $Join_{pair}$((a//b)▷(c/d))에 존재하지 않으므로 조인될 수 없다.

② 두 데이타 노드 <b, 4, 28, 30>과 <d, 9, 20, 25> begin 비교에서 28 ≮20이므로 조인될 수 없다.

③ 두 데이타 노드 <b, 4, 4, 13>과 <d, 7, 8, 10> (b.4, d.7)이 존재하고, 4<7이며 13>10이므로 조인된다.

세그먼트 조인의 이점은 각 질의 세그먼트에 나타난 모든 테이블을 다 접근할 필요없이, 1-Index와 각 질의 세그먼트의 마지막 테이블만을 접근하여 답을 구할 수 있다는 점에 있다. 즉, 디스크 접근 회수를 줄여준다. 한편, 세그먼트 조인은 두 개 이상의 질의 세그먼트로 구성된 질의에 대해서도 두 질의 세그먼트 쌍 각각에 대해 조인쌍들을 구함에 의해 쉽게 확장될 수 있다(그림 7 참조).

4. 질의처리 알고리즘

4.1. 테이블 스키마

3.1절에서 제안한 데이타 모델(XML 데이타 트리)은 엘리먼트 테이블과 텍스트 테이블로 저장된다. 엘리먼트 노드는 엘리먼트 별로, 텍스트 노드는 부모 노드의 엘리먼트 별로 나누어진다. 이 때, 엘리먼트 테이블의 스키마는 $E_{element}$(begin, end, nid, level)이고, 텍스트 테이블의 스키마는 $T_{element}$(value, begin, nid, level)이다. 그림 5는 그림 3의 데이타 모델을 저장한 테이블을 보여준다 (편의상 level은 생략하였다). E_a는 엘리먼트 노드 a들을 저장하는 엘리먼트 테이블이고, T_b는 부모가 b인 텍스트 노드들을 저장하는 텍스트 테이블이다. 이 때, 모든 테이블은 begin에 의해 오름차순으로 정렬된다.

4.2 SegmentTwig 알고리즘

세그먼트 조인은 구조 조인 조건인 조건 (4)에 nid에 의한 제약 조건 (조건 (3))이 추가된 형태이므로, 구조 조인 알고리즘인 TwigStack[10]이나 TSGeneric[13]를 기반으로 설계될 수 있다[1]. 가장 기초적인 세그먼트 조인 알고리즘은 그림 6과 같이 3단계로 구성된다.

그림 7은 단계 1을 보여준다. $q_{segment}$는 q를 구성하는 질의 세그먼트의 마지막 노드들로 이루어져 있고,

1) 이 논문에서는 TwigStack 알고리즘을 기반으로 설명한다.

begin	end	nid
1	34	1

(a)

begin	end	nid
2	15	2
4	13	4
16	33	2
18	27	4
28	30	4

(b)

begin	end	nid
3	14	3
7	11	6
17	31	3
19	26	8

(c)

begin	end	nid
5	12	5
8	10	7
20	25	9
21	23	10

(d)

value	begin	nid
t3	29	4
t2	32	2

(e)

value	begin	nid
t1	6	5
t1	9	7
t4	22	10
t3	24	9

(f)

그림 5 (a) E_a, (b) E_b, (c) E_c, (d) E_d, (e) T_b, (f) T_d

단계 1. 질의 q에서 세그먼트 분할과 1-Index 접근을 통해 $q_{segment}$를 구한다.
단계 2. TwigStack을 수행하여, $q_{segment}$를 만족하는 가지모양 데이타 노드셋을 찾는다.
단계 3. 단계 2의 결과인 가지모양 데이타 노드셋에서 각 인접 노드들이 조인쌍을 만족하는지 검사한다. 모든 인접 노드들이 조인쌍을 통과하면 최종 답이 된다.

그림 6 Naive 세그먼트 조인 알고리즘

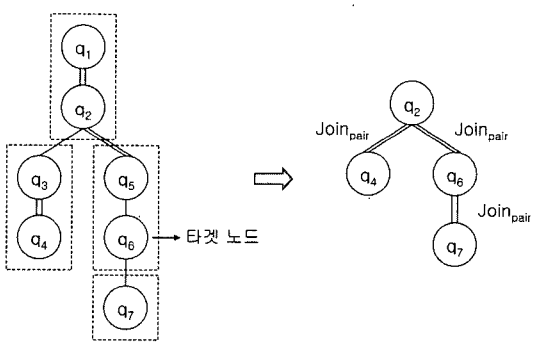

그림 7 입력 질의 q에서 qsegment 구하기 (단계 1)

각 인접 노드마다 조인쌍이 부여되어 있다. 그런 다음 $q_{segment}$에 의해 TwigStack을 수행하면, q의 결과 보다 더 많은 가지모양 노드셋이 구해진다. 왜냐하면 $q_{segment}$가 q에 비해 구조적 제약이 완화된 질의이기 때문이다. 이 때 중요한 점은 $q_{segment}$의 결과는 q의 결과를 포함한다. 마지막으로 조인쌍에 의해 각 가지모양 노드셋을 검사하면 q를 만족하는 최종 결과를 얻게 된다.

이제 SegmentTwig 알고리즘을 제안한다. 이 알고리즘의 기본적인 생각은 기초 알고리즘의 단계 3에서 수행할 nid 제약조건 검사를 가능한 일찍 함으로써 성능을 높이겠다는 것이다. 즉, 단계 2에서 단계 3을 포함하는 것이다. 그림 8은 SegmentTwig 알고리즘을 보여준다. 단계 1의 accessOneIndex 함수는 기초 알고리즘과 마찬가지로 그림 6의 과정을 수행한다. 단계 2는 앞서 설명한 대로 기초 알고리즘의 단계 2와 3이 합쳐져 있다. 다시 말해, TwigStack 알고리즘의 입출력 부분에 nid에 의한 필터를 설치한다. TwigStack 알고리즘은 엘리먼트/텍스트 테이블을 입력으로 받아 중간 결과로 경로 답안(path solution)을 구하고, 경로 답안들을 하나로 합쳐 최종적으로 가지모양 노드셋을 구하게 된다. 여기에서 테이블을 읽는 부분과 중간 결과인 경로 답안을 구하는 과정에 조인쌍을 검사하는 필터가 들어간다. 그림 8의 단계 2는 TwigStack과 비교해 함수 3개가 달라진다.

- initializeCursor : 새로이 추가
- segmentAdvance : advance를 대체
- segmentOutputPathSolutions : OutputPathSolutionsWithBlocking을 대체

```
/* 단계 1. 1-Index 접근 */
q_segment = accessOneIndex(q);
/* 단계 2. 변형된 TwigStack 수행 */
initializeCursor(q_segment);
while not end(q_segment) do
        query node n = getNext(q_segment.root);
        if not isRoot(n) then cleanStack(S_parent(n), C_n);
        if isRoot(n) or not empty(S_parent(n)) then
                cleanStack(S_n,C_n);
                if not isLeaf(n) then
                        push(S_n, C_n, pointer to top(S_parent(n)));
                else
                        segmentOutputPathSolutions(C_n);
                segmentAdvance(C_n);
end while
mergeAllPathSolutions();
```

그림 8 SegmentTwig 알고리즘

처음 두 함수는 테이블에서 튜플을 읽는 과정에 필터를 설치한 함수이다. TwigStack에서 테이블 내의 모든 튜플을 하나씩 읽어들이는 advance 함수 대신, 튜플의 nid가 조인쌍에 존재하지 않으면 그 튜플을 무시하고 다음 튜플로 진행하도록 한다. 예를 들어, {(b.4, d.7)}이라는 하나의 조인쌍만 존재할 때 b 테이블에서 nid가 5인 튜플이 입력으로 들어온다면 이 튜플은 결코 어느 것과도 조인이 될 수 없다. 그러므로, nid가 4인 튜플을 입력으로 읽을 때까지 진행한다. 마지막 segmentOutputPathSolutions 함수는 경로 답안이 구해졌을 때 인접하는 각 노드쌍들에 대해 조인쌍을 만족하는지 검사한다. 경로 답안에 모든 인접 노드쌍이 조인쌍을 만족해야만 경로 답안을 하나로 합쳐 가지 모양으로 출력하는 mergeAllPathSolutions에 전달된다.

이와 같이 TwigStack을 수정하는 과정에서 TSGeneric+ 알고리즘과 유일하게 달랐던 getNext 함수는 그대로 사용되었다. 다시 말해 각 테이블마다 XR 트리 인덱스가 부여되어 있다면 TSGeneric+ 알고리즘의 getNext 함수를 사용함에 의해 SegmentTwig 알고리즘은 동작할 수 있다.

SegmentTwig 알고리즘은 TwigStack의 정확성(correctness)을 기반으로, 입출력 부분에 조인쌍에 의한 필터를 둠으로써 정확히 가지모양 질의를 만족하는 답을 구할 수 있다. 또한, 이 과정에서 질의 세그먼트의 마지막 질의 노드를 제외한 다른 노드에 연관된 테이블들을 접근하지 않으므로 더 빠르게 질의를 처리할 수 있다.

5. 실 험

세그먼트 조인 기법의 성능을 테스트하기 위해, 인덱스되지 않은 테이블들에 대해 SegmentTwig 알고리즘과 TwigStack 알고리즘을 비교하였다. 모든 소스 코드는 C++ 언어로 작성되었고, TwigStack 알고리즘의 경우 저자[10]들이 제공해준 코드를 우리의 실험 환경에서 동작하도록 수정하였다. 실험 환경으로 1.8 GHz 펜티엄 4 프로세서, 512 MB 메모리, 40 GB 하드디스크로 구성되고, 솔라리스 8을 운영체제로 사용하는 컴퓨터에서 진행하였다.

5.1 데이타셋

실험은 하나의 인조 데이타셋과 세 개의 실제 데이타셋을 이용하여 진행되었다. XMark 데이타셋은 XMark 벤치마크 웹사이트[16]에서 제공되었고, Nasa, SwissProt, Treebank 데이타셋은 University of Washington에서 운영하는 XML 데이타 저장소[17]에서 구한 실제 데이타셋이다. 이 중 Treebank 데이타셋은 다른 데이타셋과 비해 1-Index의 크기가 대단히 크다. 각 데이타셋들의 세부 내용은 표 1에서 알 수 있다.

5.2 가지모양 질의

표 2에 보여지는 바와 같이, 실험에는 T_1부터 T_{11}까지 총 11개의 가지모양 질의가 사용되었다. 각 데이타셋마다 3개의 질의가 선택되었고, SwissProt만 2개가 선택되었다. 3개의 질의 중 첫 번째 질의는 세 개의 질의 세그먼트로 구성되었고, 두 번째 질의는 첫 번째 질의에 하나의 질의 세그먼트를 추가하였다. 그리고, 마지막 질의는 긴 질의 세그먼트 하나를 포함하고 있다. 다만, SwissProt 데이타셋은 데이타 트리의 높이가 낮아 세 번째 패턴의 질의를 선택하지 못하였다.

5.2.1 작은 1-Index(XMark, Nasa, SwissProt 데이타셋)

먼저, SegmentTwig를 실행시켰을 때, 첫 번째 단계(1-Index 접근)를 수행하는데 필요한 시간과 전체 수행 시간을 측정하였다. 표 2는 실험 결과를 보여준다. 이 세 데이타셋의 1-Index가 전부 메모리에 있고 매우 작기에, 1-Index 시간은 전체에 비해 최대 0.0003%일 정

표 1 데이타셋과 1-Index

데이타셋	엘리먼트 개수	최대깊이	크기	1-Index 노드 수	1-Index 크기
XMark	2,048,193	12	111 MB	549	34 KB
Nasa	532,963	8	23 MB	112	7 KB
SwissProt	5,166,890	5	109 MB	265	16 KB
Treebank	2,437,661	36	82 MB	338,750	20 MB

표 2 실험에 사용된 가지모양 질의

	데이타셋	질 의
T_1	XMark	closed_auction[//listitem//parlist]//author//person
T_2	XMark	closed_auction[//listitem//parlist][//author//person]//itemref//item
T_3	XMark	site//closed_auctions//closed_auction[//seller//person]//annotation//-description//parlist//listitem//text//emph//keyword
T_4	Nasa	dataset[//tableHead//title]//ingest//date
T_5	Nasa	dataset[//tableHead//title][//ingest//date]//reference//name
T_6	Nasa	dataset//reference//source//other[//author//lastName]//date//year
T_7	SwissProt	Entry[//HIV//sec_id]//Features//PEPTIDE
T_8	SwissProt	Entry[//HIV//sec_id][//Features//PEPTIDE]//Ref//Author
T_9	Treebank	SBAR//S[//PP]//VP//NP
T_{10}	Treebank	SBAR//S[//PP][//VP//NP]//ADVP//NN
T_{11}	Treebank	PP//SBAR//VP//ADVP[//NP//NN]//S//_NONE_

표 3 1-Index 시간과 총 실행시간 비교

데이타셋	질의	1-Index 시간(초)	총 실행시간(초)
XMark	T_1	0.00008	1.0
	T_2	0.00010	1.3
	T_3	0.00049	1.3
Nasa	T_4	0.00006	0.3
	T_5	0.00011	0.7
	T_6	0.00009	0.1
SwissProt	T_7	0.00008	2.3
	T_8	0.00010	5.8

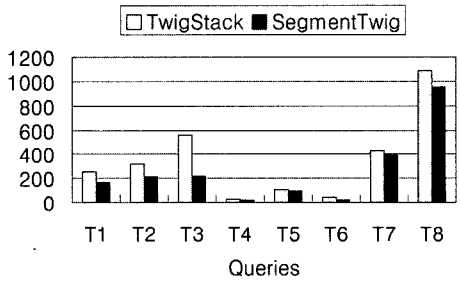

그림 9 입력 노드 수 비교 (×1000)

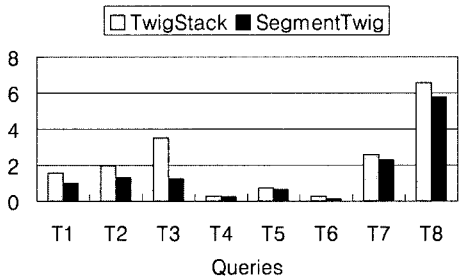

그림 10 실행시간 비교 (초)

도로 극소하였다.

다음으로, SegmentTwig 알고리즘과 TwigStack 알고리즘을 비교하였다. 그림 9는 입력 노드 수(입력 튜플 수)를, 그림 10은 총 실행시간을 보여준다. 이 두 그래프는 대단히 유사하다. 즉, 실행시간은 입력 노드 수에 비례한다. SegmentTwig는 모든 질의에 대해 입력 노드 수를 줄임에 의해 TwigStack 보다 우수한 성능을 보였다.

각 데이타셋의 첫 번째와 두 번째 질의를 비교하면, 하나의 질의 세그먼트가 더 추가됨에 따라 TwigStack은 급격히 실행시간이 증가하였지만, SegmentTwig는 T_1&T_2의 경우 증가량이 작았으며, T_4&T_5, T_7&T_8의 경우 어느 정도의 증가량을 보였다. 이는 TwigStack은 추가된 질의 세그먼트의 모든 테이블의 크기의 총합에 비례하지만, SegmentTwig는 마지막 노드에 연관된 테이블의 크기에만 영향을 받기 때문이다. 즉, T_2의 경우 마지막 노드 테이블의 크기가 크지 않지만, T_5와 T_8의 경우 마지막 노드 테이블이 크다는 것을 알 수 있다.

5.2.2 큰 1-Index (Treebank 데이타셋)

Treebank 데이타셋의 경우 1-Index의 크기가 약 20MB이므로 1-Index가 메모리에 거주하는 상황과 디스크에 저장되어 있는 두 가지 상황을 모두 가정하였다. 첫 번째 상황에서 수행되는 SegmentTwig 알고리즘을 SegmentTwig(m)이라 하고, 두 번째 상황을 SegmentTwig(d)라고 한다.

먼저, 이전 실험과 마찬가지로 SegmentTwig 알고리즘들의 1-Index 시간과 전체 시간을 측정하였다. SegmentTwig(m)의 경우 1-Index 크기 증가로 인해, 이전 실험에서 보여준 0.0003%에 비해 증가하여 1~5%에 이르렀다. 하지만, SegmentTwig(d)에서는 디스크 접근이 발생하면서 최고 42%까지 치솟았다. 당연하게도 두 SegmentTwig 알고리즘의 총 실행시간은 1-Index 시간의 격차만큼 차이났다.

다음으로, 두 SegmentTwig 알고리즘의 입력 노드수와 실행시간을 TwigStack과 비교하였다. 모든 경우

표 4 Treebank 데이타셋에 대해 1-Index 시간과 총 실행시간 비교

	SegmentTwig(m)		SegmentTwig(d)	
	1-Index 시간	총 실행시간	1-Index 시간	총 실행시간
T_9	0.07	6.14	0.85	6.86
T_{10}	0.12	3.16	0.96	4.08
T_{11}	0.09	1.61	1.10	2.60

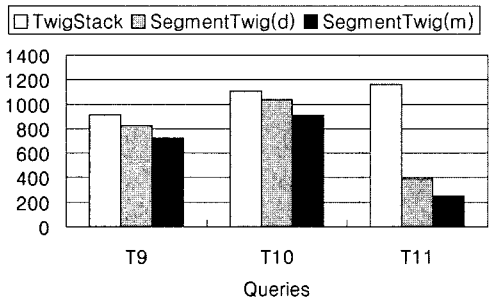

그림 11 Treebank 데이타셋에 대해입력 노드 수 비교 (×1000)

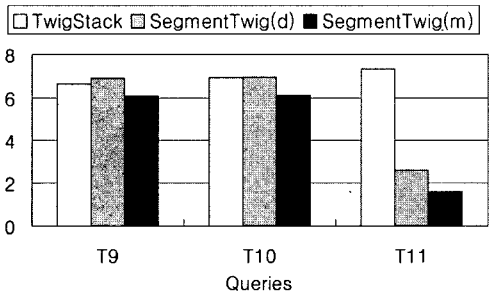

그림 12 Treebank 데이타셋에 대해 실행시간 비교 (초)

에 SegmentTwig 알고리즘이 빠르리라는 예상과는 다르게, SegmentTwig(d)는 T_9와 T_{10}에서 TwigStack 보다 느렸다. 이런 결과가 발생한 이유는 첫째, 그림 11에서 보듯이 1-Index가 디스크에 저장됨에 따라 디스크 접근양이 비슷해졌으며, 둘째, 다른 질의들의 경우 (T_1~T_8) 조인쌍들이 보통 5개 정도였음에 비해, T_9와 T_{10}에서는 약 5만개, 3만개 정도에 달했다. 즉, 하나의 노드를 읽을 때마다, 그 노드가 조인쌍을 만족하는지를 검사하기 위해 많은 비교(이진 탐색에 의한)가 발생하였다. 이것은 CPU 비용의 증가를 뜻한다. 결론적으로, 1-Index의 크기가 매우 클 경우, 디스크 접근양 뿐만이 아니라 CPU 비용도 증가할 수 있음을 보여주고 있다.

6. 결 론

이 논문에서 구조 인덱스를 구조 조인 기법을 결합한 세그먼트 조인 기법을 제안하였다. 이 기법은 질의 노드와 노드 간의 구조 조인과는 달리 구조 인덱스를 이용하여 일련의 질의 노드들을 하나의 세그먼트로 식별한 후 세그먼트와 세그먼트 사이의 조인을 수행한다. 그 결과 세그먼트마다 하나의 질의 노드만을 읽음에 의해 질의를 처리할 수 있었다. 세그먼트 조인 기법을 적용한 SegmentTwig 알고리즘은 TwigStack이나 TSGeneric~와 같은 구조 조인 알고리즘에 비해 모든 입력 테이블을 접근할 필요없이 조인트 노드, 타겟 노드, 단말 노드에 관련된 테이블만을 읽어도 원하는 결과를 구할 수 있었다. 즉, 디스크 접근 양의 감소가 성능 향상을 이끌었다.

실험 결과, 1-Index의 크기가 작을 경우 상당할 정도로 성능이 향상 되었고, 만약 질의가 더많은 질의 노드를 포함한다면 이 격차는 더 커질 것이다. 다만 1-Index의 크기가 매우 클 경우 CPU 비용의 증가가 성능에 영향을 줄 수 있음을 보였다.

참 고 문 헌

[1] R. Goldman and J. Widom, "DataGuides: Enabling Query Formulation, Optimization in Semistructured Databases," In Proceedings of VLDB Conference, pp. 436-445, 1997.

[2] T. Milo and D. Suciu, "Index Structures for Path Expressions," In Proceedings of 7th International Conference on Database Theory, pp. 277-295, 1999.

[3] J. Min, C. Chung, and K. Shim, "APEX: An Adaptive Path Index for XML Data," In Proceedings of ACM SIGMOD Conference, pp. 121-132, 2002.

[4] R. Kaushik, P. Shenoy, P. Bohannon, and E. Gudes, "Exploiting Local Similarity for Efficient Indexing of Paths in Graph Structured .Data," In Proceedings of ICDE, pp. 129-140, 2002.

[5] R. Kaushik, P. Bohannon, J. F. Naughton, and H. F. Korth, "Covering Indexes for Branching Path Queries," In Proceedings of ACM SIGMOD Conference, pp. 133-144, 2002.

[6] P. F. Dietz, "Maintaining order in a linked list," ACM Symposium on Theory of Computing, pp. 122-127, 1982.

[7] Q. Li and B. Moon, "Indexing, Querying XML Data for Path Expressions," In Proceedings of VLDB Conference, pp. 361-370, 2001.

[8] C. Zhang, J. Naughton, D. DeWitt, Q. Luo,, and G. Lohman, "On Supporting Containment Queries in Relational Database Management Systems," In Proceedings of ACM SIGMOD Conference, pp. 425-436, 2001.

[9] S. Al-Khalifa, H. V. Jagadish, N. Koudas, J. M. Patel, D. Srivastava,, and Y. Wu, "Structural

Joins: A Primitive for Efficient XML Query Pattern Matching," In Proceedings of ICDE, pp. 141-152, 2002.
[10] N. Bruno, N. Koudas, and D. Srivastava "Holistic Twig Joins: Optimal XML Pattern Matching," In Proceedings of ACM SIGMOD Conference, pp. 310-321, 2002.
[11] S. Y. Chien, Z. Vagena, D. Zhang, V. J. Tsotras, and C. Zaniolo, "Efficient Structural Joins on Indexed XML Documents," In Proceedings of VLDB Conference, pp. 263-274, 2002.
[12] H. Jiang, H. Lu, W. Wang, and B. C. Ooi, "XR-Tree: Indexing XML Data for Efficient Structural Joins," In Proceedings of ICDE, pp. 253-264, 2003.
[13] H. Jiang, W. Wang, H. Lu, and J. X. Yu, "Holistic Twig Joins on Indexed XML Documents," In Proceedings of VLDB Conference, pp. 273-284, 2003.
[14] S. Y. Chien, Z. Vagena, D. Zhang, V. Tsotras, and C. Zaniolo, "Efficient Structural Joins on Indexed XML Documents," In Proceedings of VLDB Conference, pp. 263-274, 2002.
[15] A. Berglund, S. Boag, D. Chamberlin, M. F. Fernandez, M. Kay, J. Robie, and J. Simon, "XML Path Language (XPath) 2.0 W3C Working Draft 16," World Wide Web Consortium, 2002, http://www.w3.org/TR/xpath20/
[16] A. R. Schmidt, F. Waas, M. L. Kersten, M. J. Carey, I. Manolescu, and R. Busse, "XMark: A Benchmark for XML Data Management," In Proceedings of VLDB Conference, pp. 974-985, 2002, (http://www.xml-benchmark.org)
[17] G. Miklau, University of Washington, "XML Data Repository," "http://www.cs.washington.edu/research/xmldatasets/," 2004.

배 진 욱

1996년 서울대학교 컴퓨터공학과 졸업 1998년 서울대학교 컴퓨터공학과 석사학위 취득. 1998년~현재 서울대학교 전기컴퓨터공학부 박사과정. 관심분야는 XML 데이타 처리, 문자열 데이타 처리, 시공간 데이타베이스, 데이타마이닝

문 봉 기

1996년 미국 메릴랜드 대학 졸업(박사) 현 University of Arizona, Computer Science 부교수. ACM SIGMOD (2003), EDBT (2002), ISDB (2002), WWW (2002), VLDB (2001), ACM SIGMOD (2001) 위원회 위원 역임, 2002년 ACM SIGMOD Proceedings Chair 역임. 관심분야는 XML indexing, data mining, data warehousing과 parallel & distributed processing

이 석 호

1964년 연세대학교 정치외교학과 졸업 1975년, 1979년 미국 텍사스대학교 전산학 석사와 박사학위 취득. 1979년~1982년 한국과학원 전산학과 조교수. 1982년~1986년 한국정보과학회 논문 편집위원장. 1986년~1989년 미국 IBM T.J. Watson 연구소 객원교수. 1988년~1990년 데이타베이스연구회 운영위원장. 1989~1991년 서울대학교 중앙교육연구전산원 원장. 1994년 한국정보과학회 회장. 1997년~1999년 한국학술진흥재단부설 첨단학술정보센터 소장. 1982년~현재 서울대학교 컴퓨터공학부 교수. 관심분야는 데이타베이스, 멀티미디어 데이타베이스