관계형 XML 가지 패턴 질의를 위한 비트맵 인덱스와 질의 처리 기법

(Bitmap Indexes and Query Processing Strategies for Relational XML Twig Queries)

이 경 하 † (Kyong-Ha Lee) 문 봉 기 †† (Bongki Moon) 이 규 철 ††† (Kyu-Chul Lee)

요 약 XML 데이터 량의 증가에 따라 DBMS를 이용한 XML 데이터의 저장 관리 기법들이 고안되었다. 하지만, 현재의 가지 패턴 질의 처리 알고리즘들은 XML 데이터를 태그 또는 임의 단위로 분할되고, 각 항목들이 특정 순서로 정렬된 역 리스트들을 입력으로 한다. 이러한 저장 기법의 불일치는 관계형 테이블에 나뉘어 저장되는 XML 데이터의 질의 처리에 이 알고리즘들의 적용을 어렵게 한다. 이 논문에서는 관계형 테이블에 저장된 XML 데이터에 대한 홀리스틱 가지 조인을 지원하기 위한 비트맵 인덱스와 이를 이용한 질의 처리 기법을 제안한다. 비트맵 인덱스는 많은 데이터베이스 시스템에서 지원하므로, 제안하는 인덱스와 가지 질의 처리 기법은 관계형 질의 처리 프레임워크에서 보다 이식이 용이하다. 제안하는 인덱스 기법은 압축을 통해 인덱스 크기를 줄이면서도 질의 처리시 압축해제가 불필요해 시간과 공간 효율적이다. 또한, 이 논문에서는 비트맵 인덱스만을 이용해 XML 노드들 간의 관계성을 식별함으로써, 가지 패턴 질의 처리를 레코드에 저장된 XML 데이터의 접근 없이 수행할 수 있는 혼합 인덱스를 제시한다.

키워드 : XML, 비트맵 인덱스, 가지 패턴 질의, XML 질의 처리, XML 데이터베이스

Abstract Due to an increasing volume of XML data, it is considered prudent to store XML data on an industry-strength database system instead of relying on a domain specific application or a file system. For shredded XML data stored in relational tables, however, it may not be straightforward to apply existing algorithms for twig query processing, since most of the algorithms require XML data to be accessed in a form of streams of elements grouped by their tags and sorted in a particular order. In order to support XML query processing within the common framework of relational database systems, we first propose several bitmap indexes and their strategies for supporting holistic twig joining on XML data stored in relational tables. Since bitmap indexes are well supported in most of the commercial and open-source database systems, the proposed bitmapped indexes and twig query processing strategies can be incorporated into relational query processing framework with more ease. The proposed query processing strategies are efficient in terms of both time and space, because the compressed bitmap indexes stay compressed during data access. In addition, we propose a hybrid index which computes twig query solutions with only bit-vectors, without accessing labeled XML elements stored in the relational tables.

Key words : XML, Bitmap index, Twig query, XML Query Processing, XML Database

· 본 연구는 지식경제부 및 정보통신연구진흥원의 대학 IT연구센터 육성·지원사업 (IITA-2008-C1090-0801-0031)의 연구결과로 수행되었음

† 종신회원 : 애리조나대 컴퓨터과학과 Postdoc.
bart7449@gmail.com
†† 비 회 원 : 애리조나대 컴퓨터과학과 교수
bkmoon@cs.arizona.edu
††† 종신회원 : 충남대 컴퓨터공학과 교수
kclee@cnu.ac.kr

논문접수 : 2009년 7월 6일
심사완료 : 2010년 4월 8일

Copyright©2010 한국정보과학회 : 개인 목적이나 교육 목적인 경우, 이 저작물의 전체 또는 일부에 대한 복사본 혹은 디지털 사본의 제작을 허가합니다. 이 때, 사본은 상업적 수단으로 사용할 수 없으며 첫 페이지에 본 문구와 출처를 반드시 명시해야 합니다. 이 외의 목적으로 복제, 배포, 출판, 전송 등 모든 유형의 사용행위를 하는 경우에 대하여는 사전에 허가를 얻고 비용을 지불해야 합니다.

정보과학회논문지: 데이타베이스 제37권 제3호(2010.6)

1. 서 론

XML은 단순하고 유연한 정보 표현이 가능한 이점 때문에 정보 표현과 교환의 표준으로 널리 적용되어 왔다. XML 데이터의 크기와 그 사용량이 점차 증가함에 따라 관계형 DBMS를 이용해 XML 데이터를 저장, 관리하기 위한 여러 연구가 수행되었다[1-10]. 여러 저장 기법들[9] 중 노드 단위 저장 기법[2,3,5,11]은 한 XML 노드(엘리먼트 또는 애트리뷰트)를 관계 테이블의 한 행과 대응시키는 기법으로, 많은 상용/오픈-소스 DBMS에 적용되어 왔다. 이 기법은 또한 효과적인 노드 간의 관계성 식별을 위해 레이블링 기법(labeling scheme)이나 경로 문자열(path-materialized string)[2] 등의 질의 처리 기법과 같이 이용된다. XML 질의는 일반적으로 가지 패턴(twig pattern)[12]으로 표현되는데, 이는 질의 트리 내 각 노드가 부모-자식(parent-child; P-C) 또는 조상-후손(ancestor-descendant; A-D) 관계성을 표현하는 간선(edge)으로 연결되는 모습을 보인다(그림 2 참조). 이 가지 패턴 질의를 처리하기 위한 많은 알고리즘들은 하나의 가지 패턴을 여러 개의 선형 경로(linear path)들로 분할, 처리하는 방법에 기반한다. 여기에서는 우선 각 선형 경로 패턴에 부합하는 인스턴스들을 검색하고, 이들을 조인함으로써 최종 결과를 도출한다. 예를 들어, 그림 2(a)의 가지 패턴 Q를 그림 3(b)의 Node 테이블에 저장된 그림 1에서의 XML 문서에 대해 질의하는 경우, Q는 먼저 3개의 선형 경로(linear path) //A, //A//B, //A//C로 분해된다.[1] 그리고, 각 선형 경로에 대한 인스턴스가 검색된 후, 이들을 조인하여 가지 패턴 Q에 대한 최종 결과를 구한다(이러한 기법을 구조 조인(structural join) [3]이라 한다.).

예제 1. 그림 3(b)의 XML 데이터베이스에 대해 그림 2(a)의 가지 패턴 질의를 처리하기 위한 의사 SQL 문장은 다음의 두 단계로 이루어진다.

1단계: List1 ← (select * from node N, path P where N.pid = P.pid and P.pathString like "A%" order by N.start);

List2 ← (select * from node N, path P where N.pid = P.pid and P.pathString like "B%A%" order by N.start);

List3 ← (select * from node N, path P where N.pid = P.pid and P.pathString like "C%A%" order by N.start);

2단계: select * from List1 A, List2 B, List3 C where A.start< B.start ^ A.start < C.start ^ A.end > B.end ^ A.end > C.end;

그 동안 가지 패턴 질의를 효과적으로 처리하기 위해 많은 알고리즘들이 제시되었다. 이들 알고리즘에 대한 가장 최근의 조사는 Gou 등[11]에 의해 수행되었다. 여러 알고리즘들 중 홀리스틱 가지 패턴 조인(holistic twig pattern join)[12-14]은 예제 1의 2) 단계를 처리하는데 있어 I/O 최적인 특성을 가지므로, 많은 각광을 받아왔다. 하지만, 이 가지 패턴 질의 처리 알고리즘들은 공통적으로 입력 데이터로 XML 문서 자체나 관계 테이블이 아닌, 유일 태그 이름 등의 기준으로 미리 물리적으로 분할되어 디스크에 저장된 역 리스트(inverted list)를 요구한다. 또한 각 리스트 내 항목들은 미리 정렬되어 있고, 조인 실행 시 각 리스트 내 노드들이 순서대로 읽혀져야 한다. 이런 역 리스트는 관계형 모델과는 다른 저장 방식이다. 따라서, 홀리스틱 가지 패턴 조인 알고리즘은 전 처리과정-기본 테이블로 요구 데이터들을 추출하여 역 리스트를 생성, 디스크에 다시 저장하는 과정-없이 직접 관계형 데이터베이스 시스템에 적용될 수 없다. 즉, 홀리스틱 가지 패턴 조인의 이용을 위해 테이블에서 XML 엘리먼트들을 추출, 이들을 여러 역 리스트로 분할한 후, 문서 출현 순서(document order) 또는 레이블링 기법에 의해 부여된 번호로 정렬해야 한다. 즉 예제 1의 1 단계와 같은 과정이 미리 요구된다. 하지만 많은 홀리스틱 가지 패턴 조인 기법들은 이러한 관계형 테이블에 저장된 XML 문서에 대한 지원을 하지 않아왔다.

관계형 데이터베이스에 저장된 XML 문서에 대해 홀리스틱 가지 패턴 조인을 효과적으로 지원하기 위해, 우

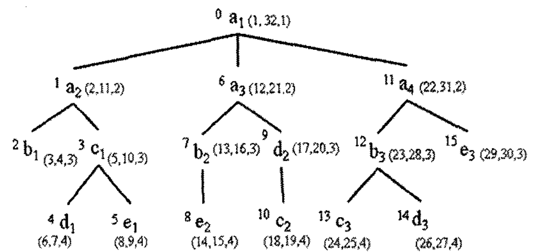


그림 1 예제 XML 문서와 구간-기반 레이블 값

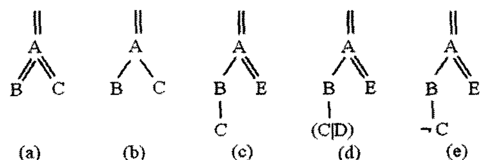
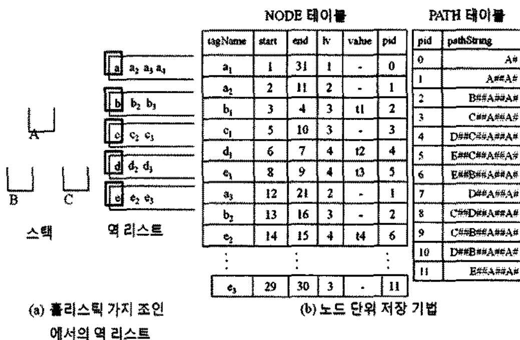


그림 2 가지 패턴 질의들

1) 이 논문에서 가지 패턴에서의 질의 노드는 A와 같이 대문자로, XML 문서에서 동일 태그 이름을 가지는 노드들은 a_1, a_2와 같이 순서 정보를 포함한 소문자로 표기한다.

리는 비트맵 기반의 인덱스와 이를 이용한 질의 처리 기법을 제안한다. **비트맵 인덱스**(bitmap index)[15]는 현재 대부분의 DBMS에서 지원되고 있으며, 이산 도메인에서의 임의 값에 대한 데이터 검색 시, 이상적인 검색 도구로 알려져 왔다. 이 논문에서는 비트맵 인덱스를 릴레이션으로 저장된 XML 엘리먼트를 태그 이름 또는 임의 기준에 따라 빠르게 접근하기 위한 수단으로 이용한다. 이를 통해 기존 홀리스틱 가지 패턴 조인 기법은 기존 관계형 데이터베이스 시스템에서 보다 쉽게 적용이 가능하다.

이 논문에서는 먼저 커서 기능이 삽입된 비트맵 인덱스를 1) **태그**, 2) **선형 경로**, 3) **(태그 이름, 레벨)**의 다양한 값-도메인에 대하여 구축한다. 그리고, 이들 인덱스를 통해 가지 질의 처리를 위해 테이블에 저장된 XML 데이터를 빠르게 접근하는 방법을 제시한다. 또한, 이 인덱스를 압축하고, 압축을 풀지 않고 질의 처리하는 방법과 여러 질의 처리 최적화 방법들도 같이 제시한다. 뿐만 아니라 이들 인덱스를 이용하여 논리 연산자를 포함한 가지 패턴 질의를 처리하는 방법을 제시한다. 마지막으로, 두 종류의 혼합(hybrid) 인덱스를 보인다. 이들 중 ***bitTwig***는 실제 XML 데이터의 접근 없이(XML 엘리먼트에 부여된 레이블 값 없이), 비트맵 인덱스 만으로 엘리먼트 간 관계성을 식별함으로써, DB로부터 추출하는 데이터 량을 최소로 하여 질의 처리 성능을 크게 향상시킨다. 마지막으로, 이들 인덱스들을 정성, 정량적으로 분석하여 장단점을 보이고, 이를 통해 사용자가 적절히 인덱스를 선택, 구축할 수 있도록 돕는다.



tagName	start	end	lv	value	pid
a_1	1	31	1	-	0
a_2	2	11	2	-	1
b_1	3	4	3	t1	2
c_1	5	10	3	-	3
d_1	6	7	4	t2	4
e_1	8	9	4	t3	5
a_3	12	21	2	-	1
b_2	13	16	3	-	2
e_2	14	15	4	t4	6
⋮	⋮				⋮
e_3	29	30	3	-	11

pid	pathString
0	A#
1	A##A#
2	B##A##A#
3	C##A##A#
4	D##C##A##A#
5	E##C##A##A#
6	E##B##A##A#
7	D##A##A#
8	C##D##A##A#
9	C##B##A##A#
10	D##B##A##A#
11	E##A##A#

그림 3 두 저장 기법의 비교

논문의 구성은 다음과 같다. 2장에서는 연구의 배경이 되는 가지 패턴 질의와 홀리스틱 가지 조인, 노드 단위 저장 기법을 살펴본다. 3장에서는 이 논문의 인덱스들과 그 구조를 소개하고, 4장에서는 비트맵 기반의 혼합 인덱스 기법을 제시한다. 5장에서는 이들 인덱스를 이용한 가지 패턴 질의의 처리 알고리즘을 기술하고, 알고리즘의 최적화와 분석을 논한다. 6장에서는 실험 결과를 논하고, 관련 연구는 7장에서, 마지막으로 8장에서 결론을 맺는다.

2. 연구 배경

2.1 가지 패턴 질의와 홀리스틱 가지 조인

가지 패턴(twig pattern)[12]은 XPath 축(axis)인 / 또는 //로 노드들이 연결되는 하나의 질의 트리이며, 하나의 XML 문서에서 여러 노드들을 선택하기 위한 선택 조건(selection predicate)이다. **가지 패턴 질의 처리**란 임의의 가지 패턴에 부합하는 모든 해를 찾는 작업이다. XML 데이터베이스 D에 대해 n개의 질의 노드로 구성되는 가지 패턴 Q의 질의 결과는 n-열의 릴레이션으로, 이는 Q에 대한 D의 유일한 매칭을 나타낸다. 가지 패턴 질의의 처리는 선형 경로의 분해와 각 선형 경로의 검색 결과를 조인하는 과정으로 구성된다. 이 때 조인 과정에서 많은 불필요한 중간 결과들이 생성되면, 질의 처리 성능을 떨어뜨린다. 이를 해결하기 위해 *twigStack*[12]을 시초로 많은 **홀리스틱 가지 패턴 조인**(holistic twig join) 기법들[13-14,16-19]이 제시되었다. 홀리스틱 가지 패턴 조인 기법은 엘리먼트간 관계성 식별을 위해 **구간 기반 레이블링 기법**(interval-based labeling scheme)[3]을 이용한다. 이 때 각 엘리먼트의 관계성은 (*startPos*, *endPos*, *levelNum*) 형식의 레이블 값으로 표현된다. 홀리스틱 가지 패턴 조인은 또한 스트림(stream)이라 불리는 커서를 가지는 역 리스트를 입력으로 한다(그림 3(a)). *twigStack*의 경우, 각 리스트는 유일한 태그 이름 별로 구분되며, 한 리스트 안의 엘리먼트들은 *startPos* 오름차순으로 정렬된다. 홀리스틱 조인 기법에서 가지 패턴 Q의 각 질의 노드 q는 하나의 스택 S_q와, 한 스트림 T_q와 연계된다. 이 알고리즘은 2 단계로 동작한다.

1) **선형 경로에 대한 질의 결과 구하기**: 가지 패턴을 여러 개의 선형 경로로 나눈 뒤, 각 선형 경로에 대한 해를 구한다. 이를 위해 질의 노드 q와 연계된 T_q로부터 엘리먼트의 레이블들을 커서를 통해 차례로 읽는다.
2) **경로 질의 결과들의 조인**: 선형 경로 해들을 다중 입력, 정렬-합병(multi-way sort-merge) 방식으로 조인한다.

*twigStack*은 조인 과정에서 선형 경로에 대한 해를 구할 때 최종 결과에 포함될 엘리먼트 만을 구할 수 있도록 보장함에 따라 I/O 최적인 특성을 보인다[20]. 어떠한 선형 경로의 해가 가지 패턴의 해를 구성하는지 검사하기 위해 *twigStack*은 질의 노드 q에 연계된 스트림 T_q에서 커서가 가리키는 XML 엘리먼트가 질의 노드 q의 조상 ancestor(q)의 스트림 $T_{ancestor(q)}$의 현재

XML 엘리먼트와 A-D(조상-후손) 관계가 있는 경우에만 T_Q의 엘리먼트를 선택한다. A-D 관계가 없는 엘리먼트들은 단순히 커서를 전진시키면서 버려진다.

기존 홀리스틱 가지 패턴 조인 기법들의 가정들 중 우리가 주목한 또 다른 문제는 조인의 결과가 구간-기반 레이블 값으로만 구성되는 n-열의 릴레이션으로, 실제 XML 데이터를 포함하지 않는다는 것이다. 즉, 조인 결과에 이름이나 텍스트 값들은 전혀 포함되어 있지 않다. 실제 XML 데이터를 출력하기 위해서는 텍스트 데이터에 접근하기 위한 레코드 주소 또는 ROWID가 요구된다. 또 다른 방법으로 그림 3(b)의 각 행, 텍스트 데이터를 포함한 한 엘리먼트의 전체 데이터를 스트림 T_Q의 각 항목으로 구성하고, 조인을 수행할 수 있으나, 이는 조인할 데이터의 크기를 크게 증가시키므로 효율적이지 않다.

2.2 관계형 DBMS에 대한 XML 저장 기법

[1]을 시작으로 XML을 관계형 데이터베이스 시스템으로 저장, 관리하기 위한 여러 기법들이 연구되었다. 가장 최근의 저장 기법들에 대한 조사는 [9]에서 수행되었다. 이 논문에서는 XML 데이터를 관계형 테이블에 저장하는데 있어 **노드 기반 저장 기법**[3]을 이용한다. 각 엘리먼트는 구간-기반 레이블링 기법으로 레이블이 부여되어 node 테이블에 한 행으로 저장된다(그림 3(b)). 테이블에 저장된 두 엘리먼트 간의 관계성은 부여된 레이블 값을 가지고 판별된다. node 테이블은 질의 처리 시 도움을 위해 경로 문자열을 저장하는 path 테이블(그림 3(b))과 조인된다. 본 논문에서는 경로 문자열의 검색 시 검색 효율성을 향상하기 위해 루트 노드부터 단말노드까지의 경로를 역으로 표현한 **역 경로**(reverse path) 문자열[5]을 이용해 각 경로 문자열들의 선택도(selectivity)를 낮춘다.

노드 기반 저장 기법은 또한 XML 데이터를 관계형 DBMS의 전통적인 인덱스 도구들을 이용해 색인할 수 있는 장점을 제공한다[5]. 만약, node 테이블이 비트맵 인덱스로 색인되어 있다면, 가지 패턴 질의 처리 시 path 테이블과의 조인은 인덱스를 이용하여 보다 빨리 처리될 수 있다. 왜냐하면, 비트맵 인덱스는 조인 인덱스로 이용될 수 있기 때문이다. 또한 비트맵 인덱스의 장점인 논리연산자의 효과적인 지원을 통해 OR, NOT 연산자를 가지는 가지 패턴을 효율적으로 지원할 수 있다. 비트맵 색인과 질의 처리를 위해 우리는 XML 엘리먼트들이 그림 3(b)와 같이 테이블에 **문서 순서**(document order)-엘리먼트의 시작 태그가 문서에 출현하는 순서-로 정렬되어 있다고 가정한다. 실제 응용에서는 튜플(tuple)이 무순서일 수 있으나, 해시 맵 등을 이용해 순서에 대한 대응 정보를 유지할 수 있으므로 이 가정은 실제 응용에 반하지 않는다.

3. 인덱스 구조

3.1 기본 비트맵 인덱스

비트맵 인덱스는 여러 개의 **비트벡터**(bit-vector)들의 집합[15]으로, 인덱스의 한 비트벡터는 어떤 값 도메인(value domain)의 한 유일 값(distinct value)에 대응한다. 이 논문에서 비트맵 인덱스의 한 비트벡터의 1들은 그 비트벡터가 대응하는 컬럼 값을 가지는 node 테이블에 저장된 엘리먼트들을 의미한다. 값-도메인은 태그 이름, 선형 경로 등이 될 수 있다. 만약 비트벡터가 태그 이름 A에 대응한다면, 이 비트벡터의 1들은 해당 릴레이션에서 태그 이름이 A인 엘리먼트들의 ROWID들을 대표한다. 따라서 태그 이름 A에 대응하는 한 비트벡터는 원 테이블을 그대로 유지하면서 가상적으로 분할된 A 태그 이름의 역 리스트와 같고, 이를 통해 홀리스틱 조인 시 테이블에 저장된 각 엘리먼트에 접근할 수 있다. 이 절에서는 먼저 기존 홀리스틱 가지 패턴 조인 기법에서 이용하는 3가지 분할 기법[12][14]을 지원하기 위한 기본 비트맵 인덱스 *bitTag*, *bitPath*, *bitTag+*를 소개한다.

bitTag는 각 비트벡터가 유일한 태그 이름에 대응하는 비트맵 인덱스이다. 이 인덱스는 그림 3에서 tagName 컬럼에 대해 색인함으로써 얻는다. 가지 패턴의 한 질의 노드에 대해 하나의 비트벡터가 질의 처리 과정 중에 요구된다. 그림 1의 XML 에 대한 *bitTag*의 예는 그림 4(a)와 같다.

bitPath는 각 비트벡터가 유일한 역 경로에 대응하는 비트맵 인덱스로 그림 3에서 pid(경로id)를 도메인으로 하여 생성된다. 이 비트벡터의 1들은 대응하는 역 경로의 단말 노드들을 나타낸다. 예를 들면, pid=2에 대응하는 비트벡터에서 1들은 pid=2인 역 경로 B/A/A/를 만족하는 B를 가리킨다. 이 인덱스는 질의 처리 과정 중에 하나의 질의 노드에 대해 여러 개의 비트벡터가 접근될 수 있다. 이는 //-축을 가진 선형 경로는 여러 경로 문자열에 대응될 수 있기 때문이다(예, 두 선형 경로 /A와 /A/A 는 //A에 대응). 그림 1의 XML에 대해 생성된 *bitPath*의 예는 그림 4(b)와 같다.

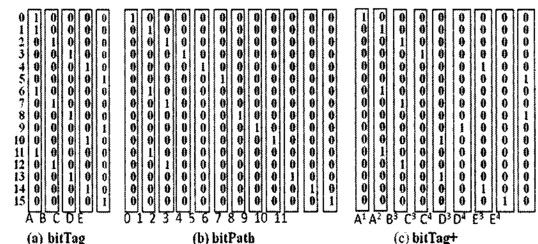


그림 4 여러 값-도메인에 대해 구성된 비트맵 인덱스들

bitTag+는 각 비트벡터가 유일한 (태그 이름, 레벨) 쌍에 대응하는 비트맵 인덱스이다. XML 데이터를 (태그 이름, 레벨)로 분할하는 기법은 *iTwigJoin*[14]에 의해 제안된 바 있다. 여기에서는 XML 엘리먼트를 먼저 태그 이름 별로 분할하고, 이들을 다시 레벨 별로 분할한다. *bitTag+*는 원 테이블은 그대로 유지하고, 인덱스를 유일한 (tagName, level) 쌍에 비트백터를 대응시킨다. 그림 1의 XML에 대한 *bitTag+*의 예는 그림 4(c)와 같다.

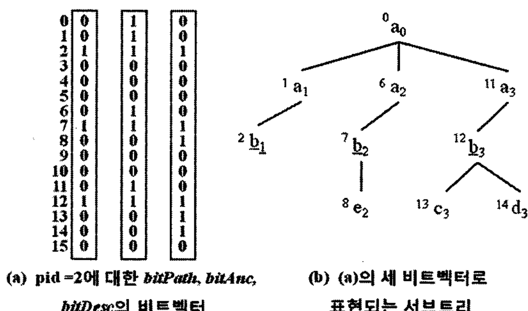
각 인덱스를 구성하는 비트벡터의 수는 애트리뷰트의 차수(cardinality)와 같다. *bitTag*는 태그 이름 별로 나뉘어진 비트벡터들을 가지며, 비트벡터의 수가 가장 적다. *bitPath*는 유일한 역 경로들에 따라 비트벡터를 가지므로 가장 많은 비트벡터를 가진다. *bitTag+*에서의 비트벡터 수는 이 둘의 중간이다. 동일한 가지 패턴 질의를 처리하는데 요구되는 비트벡터의 수는 사용되는 비트맵 인덱스의 유형에 따라 다르다. 예를 들어 그림 2(a)의 가지 패턴 질의를 처리한다고 하자. 이의 질의 노드 C에 할당되는 비트벡터는 *bitpath*의 경우 경로 //A//C에 대응하는 pid=3, 8, 9인 여러 비트벡터들이다. *bitTag*를 이용하는 경우, 태그 이름 C에 대응하는 하나의 비트벡터만이 된다. 한 질의 노드에 대해 다수의 비트백터가 할당되는 경우 이는 적용된 질의처리 기법에 따라 비트열-OR로 병합되거나 또는 비트벡터의 각 커서에 따라 1 들이 병렬로 접근될 수 있다. 이에 대한 자세한 설명은 5.3절에서 다룬다.

3.2 A-D 관계성 식별을 위한 보조 인덱스

3.1절의 기본 인덱스는 특정 태그 이름이나, 경로, 또는 태그 이름과 레벨 정보를 가지고 XML 엘리먼트들을 접근하기 위한 인덱스들이다. 엘리먼트 간의 A-D 관계성 식별의 도움을 위해 이 논문에서는 두 개의 보조 인덱스 *bitAnc*와 *bitDesc*를 제안한다.

bitAnc 인덱스는 *bitPath* 처럼 각 비트백터가 유일한 역 경로에 대응한다. 하지만, 이 인덱스에서 한 비트벡터의 1들은 대응하는 역 경로의 단말 노드를 포함하여 경로 상의 모든 조상 노드들을 대표한다. 즉, *bitAnc*의 한 비트벡터는 그 비트벡터가 대응하는 한 경로를 구성하는 모든 XML 엘리먼트들을 대표한다. 예를 들어, 그림 5(a)의 *bitPath*의 비트벡터에서 위치 2에 존재하는 1은 선형 경로 /A/A/B의 단말 노드 b_1을 나타낸다. 반면에 bitAnc에서 위치 0과 1에서의 1들은 b_1의 조상 노드인 a_0와 a_1을 나타낸다.

bitDesc 인덱스는 *bitAnc*와 반대로 각 비트벡터는 대응하는 선형 경로 상의 단말 노드에 대하여 단말 노드를 포함하여, 그 후손 노드 모두를 대표한다. 그림 5(a)의 *bitDesc* 비트벡터는 경로 /A/A/B의 단말 노드와 이의 후손 노드 모두를 대표한다. 그림 5(b)는 그림 5(a)의 세 비트 벡터가 대표하는 부분 트리를 보인다. 이는 임의의 한 경로가 주어졌을 때 이들 보조 인덱스를 이용하여 해당 선형 경로 상의 조상 노드들과 후손 노드들을 구분할 수 있음을 보인다.



(a) pid =2에 대한 *bitPath*, *bitAnc*, *bitDesc*의 비트벡터 (b) (a)의 세 비트벡터로 표현되는 서브트리

그림 5 보조 인덱스들

3.3 인덱스 압축

노드 기반 저장 기법에서 릴레이션의 튜플 수는 XML 노드(엘리먼트 또는 애트리뷰트)의 수와 동일하며, 엘리먼트의 수가 증가할수록, 비트벡터의 길이 l도 길어진다. 또한, 한 인덱스 내 비트벡터의 수 n은 애트리뷰트의 차수(cardinality)에 비례하므로, 한 비트맵 인덱스의 크기는 $n \times l$비트로, 인덱스가 원 데이터보다도 커질 수 있다. 이런 경우 압축기법이 인덱스 크기를 줄이는데 효과적이다.

이 논문에서는 비트벡터를 압축하는데 있어 **런-길이 인코딩**(run-length encoding) 기법의 일종인 WAH (Word-Aligned Hybrid)[21]을 이용한다. 하나의 런은 동일한 비트가 연속하는 순열(sequence)을 의미하며, 이 순열의 길이를 나타내는 보다 적은 비트들로 대체하여 압축한다. WAH는 비트들을 워드 단위로 압축하는데, 압축되지 않은 비트들을 가지는 리터럴 워드와 압축된 비트들로 구성된 압축 워드가 혼재된 형태를 취한다. 이 기법은 비트열-논리 연산이 실제 컴퓨터에서는 워드 단위로 수행되는 사실 때문에 비트열 연산을 압축된 상태에서도 빠르게 수행할 수 있는 이점을 가진다[21]. 그림 6은 8,000 비트로 구성된 비트가 어떻게 4 개의 워드

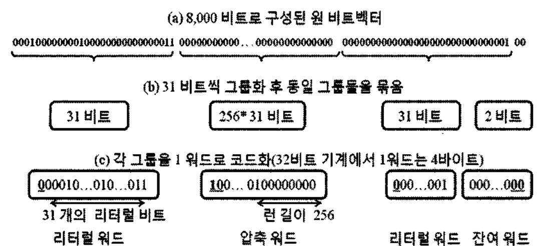


그림 6 32-bit 머신에서 비트벡터의 압축

(32bit*4=128 bit)로 압축되는지 보인다. 각 워드의 첫 번째 비트는 그 워드가 압축되었는지 아닌지를 표시한다. 첫 비트가 1이면, 그 워드는 두 번째 비트가 표시하는 비트의 연속적인 순열을 나머지 30개의 비트들이 가리키는 길이만큼 압축한다. 예를 들어, 그림 6 하단의 두 번째 워드는 압축 워드(첫 번째 비트가 1)로 256개의 0(두 번째 비트가 0)을 압축한다. 리터럴 워드의 경우는 원래 비트들을 그대로 31개씩 보유한다. 그림의 네 번째 워드는 비트들을 31개씩 그룹화하고 남은 잔여 비트들을 가진다. 잔여 비트들의 수는 그림에는 보이지 않은 다른 워드에 저장한다. 비트벡터의 수가 많을수록 비트벡터 내 1의 숫자는 적어져(sparse), 비트벡터는 더 많이 압축된다. 실험에서 압축된 인덱스는 크기 면에서 구간-기반 레이블보다도 훨씬 작음을 보인다(표 4).

3.4 압축된 비트벡터에서의 커서 이동

비트맵 인덱스에서 각 비트벡터의 1들은 특정 값을 가지는, 테이블에 저장된 XML 엘리먼트들을 대표한다. 홀리스틱 가지 패턴 조인에서는 역 리스트 내 각 엘리먼트를 커서를 이용해 차례대로 방문한다. 이를 위해 우리는 **커서 기능이 부여된 비트벡터**(cursor-enabled bit-vector)를 고안하였다. 비트벡터의 각 1들을 차례로 접근하며, 해당 비트의 위치 값(ROWID)를 차례로 출력한다. WAH기법으로 압축된 비트벡터 상에서 압축을 해제하지 않으면서 커서 이동이 가능하도록 커서 C는 아래와 같이 3개의 물리적 위치와 1개의 논리적 위치를 가진다.

- **C.position**: 압축되지 않은 비트벡터에서의 해당 비트의 위치(논리적 위치 값)
- **C.word**: 커서가 방문 중인 현재 워드
- **C.bit**: 커서가 방문 중인 현재 워드에서의 비트 위치
- **C.rest**: 만약 커서가 잔여 워드의 비트를 가리킬 경우 해당 비트의 위치

이 커서 C를 이용하면, 연속적으로 반복하는 0을 매번 커서로 방문하는 비효율성을 줄일 수 있다. 런-길이 인코딩에서 연속하는 0들은 하나의 워드로 압축되므로 압축 워드 1 개를 읽는 것으로 무수히 많은 0들을 건너뛰는 것과 같은 효과를 가진다. 알고리즘 1은 커서의 현재 위치에서 가장 가까운 1로 커서를 이동시키고 해당 1의 위치 값을 반환하는 함수 getPosOfNext()을 설명한다. 만약 현재 방문 중인 워드에 1이 없으면, 해당 워드는 그냥 통과된다(3, 16줄). 커서는 현재 위치부터 다음의 1을 찾으므로 이 함수의 복잡도는 압축된 비트 벡터를 표시하는데 들인 워드의 수로 제한된다.

```
   입력: 비트벡터 v
   출력: 다음 1의 위치 값
1  while C.word < v 내 워드의 수 do
2    if 현재 방문 중인 워드가 리터럴 워드 then
3        if 워드의 모든 비트가 0 then
4            C.position+=31;
5        else if 모든 비트가 1 ∧ C.bi < 워드 내
   비트 수
6        then
7        C.bit++; return C.position++;
8        else
9            while C 가 워드 내 비트를 가리키는
10 동안 do
11               if C가 가리키는 비트가 1 then
12           return C.position;
13               C.bit++; C.position++;
14   else  //현재 워드가 압축 워드
15       if 두번째 비트가 1 then // 워드가 1 들로
16 채워짐
17           while C 가 현재 런에 있는 동안
               C.bit++; return C.position++;
18       else // 워드가 0 으로 채워진 경우
19           C.bit+=run_length*31;
20       C.position+=run_length*31;
21   C.word++; C.bit ←0;
   if 잔여 워드 존재  ∧ C.word 가 이 워드를 가리킴
   then
       if C.rest 가 가리키는 비트가 1 then
           return C.position
     C.rest++;  C.position++;
```

알고리즘 1 getPosOfNext1(v)

```
   입력: 비트벡터 v 와 커서의 논리적 위치 값 pos
   출력: 위치 값 pos 에 위치한 비트의 부울 값
1   if pos< C.position then
2       if pos > B. position then B ← C; //B: 역방향
3  커서
4       return getBooleanAtBkd(v, pos);
5  while C.word ≠ v 의 워드 수 do
6      distance ← pos – C.position;
7    if 방문 중인 워드가 압축되었는가 then
8      if distance < run_length* 31 then
9         C.position++; C.bit++
10        return 두번째 비트의 부울 값
       else C.position+=run_length*31; c.bit ← 0;
11     C.word++;
12   else
13     if distance < 워드 내 잔여 비트의 수 then
14         C.position++; C.bit++;
15         return distance-번째 비트의 부울 값
16     else C.position++; C.bit ← 0; C.word++;
   if 잔여 워드 존재 ∧ distance < 잔여 워드의 비트 수
17 then
18   C.rest++; C.position++;
19   return  distance-번째 비트의 부울 값
   return error //C 가 End of Vector 를 만나는 경우
```

알고리즘 2 getBooleanAt(v, pos)

예제 2. 커서 C가 그림 6 하단의 첫 워드의 31번째 비트를 가리킨다면, C의 현재 상태는 {31, 0, 31, 0}이다. 다음 1의 위치를 구하기 위해 커서는 두 번째 워드의 첫 비트부터 조사한다. 이 워드는 0들의 순열을 압축했으므로, C에 런 길이*31을 더함으로써 비트 단위 방문을 피한다. 따라서 다음 1을 가리키는 커서 C의 상태

는 {7998, 1, 31, 0}이 된다(7998=31+256 *31+31).

또한 ***bitTwig***에서 A-D 관계성 식별을 위해 주어진 위치에 존재하는 비트의 부울 값을 반환하는 함수(알고리즘 2)를 이용한다. 이 함수를 위해 우리는 순방향 커서 C와 역방향 커서 B를 *bitAnc* 인덱스의 비트벡터에 추가한다(다른 인덱스들의 비트벡터에는 순방향 커서 C만 존재한다). 커서를 2개를 두는 이유는 커서 1개를 두는 경우 커서가 역방향으로 이동하였다 다시 순방향으로 이동하는 경우 이동 거리가 보다 길어지기 때문이다.

이 함수에서 만약 주어진 위치 값 pos가 순방향 커서 C의 논리적 위치 값보다 작다면, 역방향 커서 B 가지고 동작하는 쌍둥이 함수 getBooleanAtBkd()를 호출한다(1번째 줄). 이를 통해 커서 C가 앞뒤로 움직이는 일을 방지한다. 만약 커서 이동이 순방향이면, 먼저 커서 C가 움직여야 할 논리적 거리를 계산한다(5번째 줄). 이때의 커서 이동 또한 압축 워드로 인해 단축될 수 있다(6-7번째 줄). 커서가 한 워드를 지날 때마다 거리는 31의 배수로 증가한다. 이 논문에서 getPosNext1과 getBooleanAt 함수는 단일 비트벡터에 대해 절대 같이 호출되지 않는다. 기본 인덱스의 비트벡터들은 getPosNext1에 의해서만 접근되고, *bitAnc*의 비트벡터만 getBooleanAt에 의해서만 접근된다.

예제 3. 커서 C가 현재 그림 6 하단의 첫 번째 워드의 16번째 비트를 가리키고 있다고 하자(C={16, 0, 16, 0}). 이때 3,000번째 비트의 부울 값을 얻기 위해 잔여 이동 거리를 계산한다(2884=300-16). 다음으로 현재 워드가 리터럴 워드이고, 잔여 거리가 워드 내 비트 수보다 작지 않으므로 커서를 1워드 전진시킨다. 그러면, C.position은 31이 되고 잔여 거리는 2,869(=3000-31)이 된다. 두 번째 워드는 압축 워드이고, 잔여 거리 < run_length*31)이므로, 이 워드의 두 번째 비트 0을 3,000번째 비트의 부울 값으로 출력한다.

3.5 인덱스의 특성

이 절에서는 앞서 소개한 비트맵 인덱스들의 주요 특성들을 설명한다. 비트열-AND, OR와 같은 비트열 논리 연산자는 각각 ⊙, ∨으로 표기한다. *bitTag*에서 태그 이름 a와 대응하는 비트벡터는 bitTag(a), *bitPath*에서 pid i에 대응하는 비트벡터는 bitPath(i), *bitTag+*에서 태그 이름 a와 레벨 l에 대응하는 비트벡터는 $\text{bitTag}^{+}(a,l)$로 표기한다. 또한, *bitAnc*와 *bitDesc*에서 pid i에 대응하는 비트벡터는 각각 bitAnc(i)와 bitDesc(i)로 표기한다.

보조정리 1. 이 논문에서의 비트맵 인덱스의 비트벡터들은 다음과 같은 특성을 가진다.

$$\text{bitTag}(a) \equiv \bigvee_{\text{level } l=1}^{\text{height(doc)}} \text{bitTag}^{+}(a,l) \equiv \bigvee_{i \in \text{end}(a)} \text{bitPath}(i) \quad (1)$$

$$\bigvee_{i \in \text{pids}(s)} \text{bitPath}(i) \equiv \partial_{\text{path}=s,}(\text{bitTag}(a)), \quad \text{leaf}(s) = a \quad (2)$$

$$\text{bitAnc}(i) \odot \text{bitPath}(i) \equiv \text{bitDesc}(i) \odot \text{bitPath}(i) \equiv \text{bitPath}(i) \quad (3)$$

$$\left(\bigvee_{i \in \text{pids}(s)} \text{bitDesc}(i)\right) \odot \text{bitTag}(a) \equiv \partial_{a \in \Delta_{\text{pids}(s)}}(\text{bitTag}(a)), \quad \text{leaf}(s) = a \quad (4)$$

여기에서, Δ_i는 원 XML 문서에서 pid i에 대응하는 태그들을 루트로 하는 부분 트리를 의미한다. end(a)는 태그 이름 a로 끝나는 선형 경로들의 pid를, leaf(s)는 선형 경로s의 단말 노드의 태그 이름을, pids(s)는 경로 문자열 s에 대응하는 pid들을 반환하는 함수들이다.

식 (1)은 복수 개의 입력 비트벡터가 홀리스틱 가지 조인을 위한 태그 기반의 단일 비트벡터로 병합될 수 있음을 보인다. 식 (2)는 주어진 경로 표현식을 만족하는 *bitPath*의 비트벡터들을 병합한 결과는 해당 경로 표현식의 단말 노드에 대한 *bitTag*의 비트벡터 중 그 경로 표현식을 만족하는 1들만 선택한 결과와 동일함을 보인다. 식 (3)은 *bitAnc*와 *bitDesc*는 대응하는 경로의 단말 노드를 포함하며, 또한 *bitPath*를 이용해 *bitAnc*와 *bitDesc*의 비트벡터에서 단말 노드를 대표하는 비트들을 선택할 수 있음을 보인다. 식 (4)는 경로 표현식 s의 단말 노드를 루트로 하는 부분 트리에서 이 트리에 속하는 노드들을 비트벡터들을 이용해 선택할 수 있음을 보인다.

4. 인덱스만을 이용한 엘리먼트 간 관계성 식별

홀리스틱 가지 조인을 통해 생성되는 가지 패턴의 해는 레이블(*startPos*, *endPos*, *levelNum*)으로 구성되는 n-열의 릴레이션이다. 하지만, 실제 가지 패턴 질의의 해에서는 태그 이름이나 텍스트 값들과 같은 다른 정보들 또한 포함되어야 하며, 이를 위해 ROWID들이 최종 결과를 구성하기 위해 요구된다. 가지 패턴 질의 처리를 위해 3장의 기본 인덱스들은 ROWID를 제공하지만, 질의 처리 시 엘리먼트간 관계성 식별을 위해 레이블 값을 계속 필요로 한다. 이 장에서는 레이블링 기법의 도움 없이 비트맵 인덱스만으로 가지 패턴 질의에 대한 해를 계산함으로써 I/O 비용을 줄이는 방법을 제시한다. ***bitTwig***로 명명한 이 혼합 인덱스는 2종류의 비트맵 인덱스를 가지고 가지 패턴 질의를 지원한다. 이 기법은 우리의 가정에서 XML 엘리먼트들이 테이블에 문서 출현 순서대로 정렬되므로, 이것이 레이블 값의 *startPos*를 대체할 수 있다는 점을 이용한다. 또한, *bitAnc* 인덱스가 *bitPath* 인덱스가 설명하는 모든 선형 경로에 대해 그들의 조상 노드들을 대표하므로 이를 이용해

XML 엘리먼트 간의 A-D 관계성을 식별할 수 있다. 이 기법을 이용한 가지 패턴 질의의 결과는 자체적으로 가지 패턴의 해에 대한 ROWID들을 갖게 된다.

어떠한 두 엘리먼트의 문서에서의 선행(preceding), 후행(following) 관계는 *bitPath*의 비트벡터에서 대응하는 1들의 위치 값을 비교함으로써 간단하게 파악할 수 있다. 또한, A-D와 P-C 관계성 식별을 위해 *bitAnc* ∪ *bitPath*를 이용한다. 인덱스_이름(v)와 인덱스_이름(v)[k]를 각각 애트리뷰트 값 v에 대한 임의 인덱스의 비트벡터와 해당 비트벡터에서의 k번째 비트라 하자. 엘리먼트 a에 대응하는 경로 id는 pathid(a)라 하고, pos(a)는 엘리먼트 a의 문서 출현 순서라 한다.

정리 1. 두 엘리먼트 a, d가 다음 모든 조건들을 만족하면, a는 d의 조상이다.

1. $\mathrm{pathid(a)} \neq \mathrm{pathid(d)}$,
2. $\mathrm{pos(a)} < pos(\mathrm{d})$,
3. $\mathrm{bitAnc}(\mathrm{pathid(d)})[\mathrm{pos(a)}] = 1$,
4. $\mathrm{bitPath}(\mathrm{pathid(a)})[\mathrm{k}] = 0, \forall \mathrm{k}: \mathrm{pos(a)} < k < pos(\mathrm{b})$.

대우 증명(proof by contraposition)을 통해 이 정리를 증명한다.

증명. 정리 1의 대우는 "만약 a가 d의 조상이 아니면, 두 엘리먼트는 위 조건들 최소1개는 만족하지 않는다"이다. 이 대우가 사실이면, 위 정리 또한 사실이다. (1)의 조건이 사실이 아니면, a는 d의 조상이 될 수 없다. 왜냐하면, 두 엘리먼트의 경로가 같기 때문이다. (2) 문서 출현 순서(document order)에서 a의 시작 태그가 d보다 나중에 나타나는 경우에는 pos(a) > pos(d)이다. 조상 엘리먼트의 시작 태그는 언제나 후손 노드보다 먼저 출현해야 하므로, 이 경우 a는 d의 조상이 될 수 없다. (3) 만약 a가 선형 경로pathid(d) 상의 조상 노드가 아니라면, a에서 pos(a) 위치의 비트는 1이 아니다. 왜냐하면, a는 선형 경로 pathid(d) 를 만족하는 단말 노드와 그들의 조상 노드들을 포함하기 때문이다. (4) 만약 비트벡터 bitPath(i)에서 pos(a)와 pos(d) 구간에 1이 존재한다면, d에 선행하는 선형 경로 bitPath(i)의 다른 인스턴스가 존재한다. 이 경우, 경로 pathid(j)의 단말 노드는 언제나 경로 pathid(i)의 다음 인스턴스에 선행하여 존재하고 pathid(j)는 pathid(i)의 확장이기 때문에, d는 더 이상 a 의 후손이 될 수 없다. □

그림 2(a)와 (b)의 가지 패턴을 처리하기 위한 *bitAnc* ∪ *bitPath*의 예는 그림 7에서 보인다. 그림 7에서 각 비트벡터 밑의 기호는 해당 비트벡터가 대표하는 태그 이름과, pid, 레벨 값을 의미한다(예, B(2,3)은 pid=2, level=3인 B 태그). 이 그림에서는 이해를 돕기 위해 *bitPath*로부터 온 비트는 밑줄을 침으로써(1), 두 비트벡터를 하나로 합쳐서 보였다. *bitTwig*에서 두 비트벡터를 실제로 병합하지는 않는다.

	B(2, 3)	A(0, 1)	A(1, 2)	C(3, 3)	C(8, 4)	C(9, 4)
0	1	1	1	1	1	1
1	1	0	1	1	0	0
2	1	0	0	0	0	0
3	0	0	0	1	0	0
4	0	0	0	0	0	0
5	0	0	0	0	0	0
6	1	0	1	0	1	0
7	1	0	0	0	0	0
8	0	0	0	0	0	0
9	0	0	0	0	1	0
10	0	0	0	0	1	0
11	1	0	1	0	0	1
12	1	0	0	0	0	1
13	0	0	0	0	0	1
14	0	0	0	0	0	0
15	0	0	0	0	0	0

그림 7 관계성 식별을 위한 두 비트벡터의 이용

예제 4. 그림 7의 비트벡터들을 가지고, 그림 1의 a_2와 c_1, 그리고, a_2와 c_2 간의 A-D 관계성을 검사한다. 여기에서 a_2와 c_1, c_2의 위치 값은 그림 1에서 보이는 바 같이 각각 1,3,10이다. 정리 2.2에 따라, a_2와 c_1에는 A-D 관계성이 존재하나, a_2와 c_2간에는 A-D 관계성이 존재하지 않는다. 왜냐하면, c_1과 a_2는 정리 1의 모든 조건을 만족하지만, c_2의 경우 bitAnc(pathid(c_2))[pos(a_2)]가 0이기 때문이다. 이는 정리 1의 조건 (3)를 위배한다.

예제 5. 그림 7의 비트벡터들을 가지고, 각각 위치 값이 1, 7인 a_2와 b_2 간에는 A-D 관계성이 존재하지 않음을 확인할 수 있다. 왜냐하면, a_2에 대해 bitPath(1)[k] = 1,3k:pos(a_2)k$<$pos(b_2)으로, 정리 1의 조건 (4)를 위배한다.

엘리먼트 a_i가 어떤 경로 문자열에 대응하는 i번째 엘리먼트이고, 엘리먼트 d 의 조상이라면, 정리 1의 2와 4에 따라 $\mathrm{pos}(a_i) < \mathrm{pos(d)} < \mathrm{pos}(a_{i+1})$의 특성을 가진다. 이는 해당 구간에 대해 1이 존재하는지를 구간 검색을 수행하게 하는데, 이를 단순히 한 커서를 이동함으로써 각 비트를 매번 비교하고자 하면, 성능 저하가 있다. 이런 구간 검색을 없애기 위해, 커서 위치 다음의 1을 미리 보기(look ahead)(이에 대해서는 5.3절에 설명한다)와 정리 1의 3를 먼저 검사함으로써 구간 검색을 피하는 방법을 이용한다. 이는 다음과 같이 동작한다. 만약 엘리먼트 d 에 대한 비트벡터 bitAnc(d)에서 pos(a_i) 위치의 비트가 0이면, 정리 1.(3)에 따라, a_i는 d의 조상이 어차피 아니므로 d가 속한 경로문자열 j에 대한 bitPath(j)에서 구간 pos(a_i)와 pos(d_{i+1})에서 1이 있는지 비교를 다시 할 필요가 없다.

알고리즘 3은 두 엘리먼트 간의 A-D 관계성을 정리 2에 따라 두 비트맵 인덱스를 가지고 검사하기 위한 함수를 보인다. 1-2 번째 줄에서, 이 함수는 먼저 두 엘리먼트가 하나의 질의 노드에 대응하면서, 경로 id가 같은지 먼저 비교한다. 만약 사실이면, 두 엘리먼트는 동일한 선형 경로에 대한 서로 다른 인스턴스들의 단말 노

드들이므로, 둘 간에 A-D 관계성이 없다. 하지만, 만약 a와 d가 하나의 질의 노드에 대응한다 하더라도 서로 다른 두 비트벡터에 의해 대표된다면, 둘 간에는 A-D 관계성이 존재할 수 있다(예를 들면, 경로 질의 //A에 대해 a_1이 경로 /A의 인스턴스이고, a_2가 경로 /A/A의 인스턴스인 경우). bitTwig 기법에서는 비트벡터들을 병합하지 않으므로, 어떠한 비트벡터가 현재 노드에 대응하는지를 파악할 수 있다. 3, 4, 5번째 줄은 각각 정리 2의 2, 3, 4번째 규칙을 검사한다. 알고리즘에서 $Curr_a$는 질의 노드 a에 대해 현재 비트벡터의 커서 값을 기록하는 변수이며, $Next1_a$는 미리 보기로 알아내는 커서의 현재 위치의 다음에 나타나는 1의 위치를 기록하는 변수이다. 우리는 이를 이용해 정리 1. (4)에서의 구간 검색을 회피한다(5.3 절 참조)

```
     입력: 두 엘리먼트 a, d
     출력: true or false
 1   if Curr_a.pid ≡ Curr_d.pid then
 2       RETURN false;
 3   else if pos(a) < pos(d) then
 4       if getValueFromBitAnd(Curr_a.pid, Curr_d.pos) then
 5           if Next1_a[Curr_a.pid].pos < Curr_d.pos
 6               then return false;
 7           else return true;
 8       else return false;
 9   else return false;
10   end
     Function getValueFromBitAnd(pid, pos)
11       return pid 를 대표하는 bitAnc 의
12   비트벡터에서 pos 위치의 값을 반환; //
13   getBooleanAt 함수 참조
     end
```

알고리즘 3 checkAD(x, y)

P-C(부모-자식) 관계성 검사는 위의 A-D 관계성 검사와 더불어 $Level(a_i) = Level(d_j)+1$인 조건을 추가로 만족하는지를 검사함으로써 수행된다. 이를 위해 각 비트벡터의 헤더에 레벨 정보를 추가로 기입하고 이를 P-C 관계성 검사에 이용한다.

5. 가지 패턴 질의 처리

5.1 알고리즘의 개요

알고리즘 4는 비트맵 인덱스를 이용한 홀리스틱 가지 패턴 조인을 위한 이 논문의 전체 알고리즘을 보인다. 여기에서 입력은 선택된 인덱스 기법에 따라 달라질 수 있다.

주어진 가지 패턴 질의 Q에 대해 먼저 가능하다면, 가지치기(pruning)을 통하여 가지 패턴 내 질의 노드의 수를 줄인다(1번째 줄). 이는 질의 처리에 참여할 질의 노드의 수를 줄이는 효과와 동시에 가지 패턴 조인 시

```
 1   가능한 경우 가지 패턴 Q 를 Q' 로 가지치기;
     //5.4 절 참조
 2   각 질의 노드에 비트벡터(들)을 할당; // 5.2 절 참조
 3   while ¬end(root) do
 4       q ← getNext(root);
 5       if q ≠ root then
 6               cleanstack(S_parent(q), Curr_q.pos);
 7       if q ≡ root ∨ ¬empty(S_parent(q)) then
 8        cleanstack(S_q, Curr_q.pos);
 9        커서 Curr_q 와 Next1_q[Curr_q.pid] 를 스택 S_q 에
     삽입; // 5.3 절 참조
10        advance(q); //알고리즘 3 참조
11        if isLeaf(q) then
12            showSolutionFromStacks(q); //twigStack[12]
     참조
13            pop(S_q);
14        else advance(q, Curr_parent(q).pos); //알고리즘 5
     참조
15   mergeAllPathSolutions(); //twigStack[12] 참조
16   Function getNext(q)
17      if isLeaf(q) then return q;
18      foreach q_i ∈ children(q) do
19         n_i ← getNext(q_i);
20         if ¬exist(q_min) ∨(curr_ni.pos < Curr_qmin.pos) then
     q_min ← n_i;
21         if ¬exist(q_max) ∨(curr_ni.pos > Curr_qmax.pos) then
     q_max ← n_i;
22         if parent(n_i) ≠ q then return n_i;
23      while checkAD(q, q_max) //알고리즘 1 참조;
24      do advance(q);
25      if Curr_q.pos > Curr_qmin.pos then return q_min;
26      else return q;
27   end
```

알고리즘 4 가지 패턴 질의 처리 알고리즘

읽어야 할 데이터의 양도 줄인다. 다음으로, 가지 패턴의 각 질의 노드에 사용자가 정한 우리의 인덱스 기법에 따라 적절한 비트벡터(들)을 할당한다(2 번째 줄), 그리고 나서, 3-14 줄까지의 과정을 루트 질의 노드가 더 이상의 입력이 없을 때까지, 즉 루트 질의 노드에 할당된 모든 비트벡터들의 커서가 EOV(End Of bitVector)에 도달할 때까지 반복한다. getNext(q) 함수는 가지 패턴 Q에서 질의 노드 q의 후손 노드 중 현재 커서의 위치($Curr_{nj}$·pos)가 가장 작은 질의 노드 n_i를 반환하는 함수이며, CleanStack(S_q, pos)는 pos 위치의 비트가 대표하는 엘리먼트의 조상이 아닌 엘리먼트들은 스택으로부터 꺼내는 함수이다. 이 두 함수에서 A-D 관계성을 식별하기 위해, 우리는 선택된 인덱스 기법에 따라 구간-기반 레이블링 기법으로 부여된 레이블을 이용하거나 또는 알고리즘 3의 checkAD(x,y)를 이용한다. 함수 advance(q)는 질의 노드 q에 할당된 비트벡터(들)의 커서(들)을 다음의 1로 전진시키고, 이 1의 위치, 즉 커서가 새로 가리키는 위치 값을 커서 $Curr_q$.pos 변수로 할당한다. 이에 대해서는 5.3절에서 다시 자세히 설명한다. 우리는 질의 처리 시 불필요한 노드에 대한 레이블

값을 읽는 것을 방지하기 위해 advance(q, k)라는 기법을 추가로 이용한다. 이는 5.4절에서 설명한다. showSolutionFromStack() 함수는 스택에 적재된 노드들을 서로 관계성에 따라 연결시켜 선형 경로의 해를 출력한다.

mergeAllPathSolutions()함수는 각각 구해진 선형 경로의 해를 정렬-합병 조인의 방식으로 합병을 수행한다. 표 1은 이 논문에서 지원하는 비트맵 인덱스를 이용한 여러 가지 패턴 질의 기법과 이 기법에서 이용하는 비트맵 인덱스의 유형들을 정리한 것이다.

5.2 비트벡터의 선정

가지 패턴 질의 처리 시 한 질의 노드에 할당되는 비트벡터의 수는 선택된 질의 처리 기법에 따라 다양하다. 표 1에서 정리한 바와 같이, bitTag 기법에서는 각 질의 노드에 하나의 비트벡터를 할당한다. bitPath 기법에서는 어떠한 경로 표현식 s를 만족하는 여러 비트벡터들이 정리 1.2에 따라 병합되어 질의 노드 q에 할당된다. descTag기법에서는 가지 패턴 Q의 각 노드 q에 대해 대응하는 경로 표현식 s를 구한 후, 이를 만족하는 비트벡터 bitDesc(pids(s))들을 구하고, 이들을 모두 병합한다. 다음으로 해당 질의 노드 q에 대한 비트벡터 bitTag(a)를 구하고 이를 병합된 *bitDesc*의 비트벡터와 비트열-AND 연산으로 병합하여 q에 할당한다. descTag와 bitTwig 기법은 한 질의 노드에 여러 비트벡터를 할당하며, 할당된 비트벡터들은 병합되지 않는다. descTag 기법의 경우, 질의 노드 q에 대하여 q의 태그 이름을 가지면서, q의 조상 노드 ancestor(q) 또는 부모 노드 parent(q)에 대하여 다음의 레벨 조건을 만족하는 비트벡터들만을 가져온다. 이는 병합할 비트벡터의 수를 줄이는 효과가 있다.

(1) ancestor(q) 의 비트벡터들 중 최소 레벨 값보다 레벨 값이 큰 비트벡터들

(2) parent(q)의 각 비트벡터들에 대해 레벨 값 + 1 을 레벨로 하는 비트벡터

bitTwig 기법에서는 항상 bitAnc(i)와 bitPath(i)의 쌍(들)을 한 질의 노드에 할당한다.

비트 벡터를 질의 노드에 할당할 때, 만약 가지 패턴 Q의 어떠한 질의 노드가 비트벡터를 할당 받지 못한다면, 그것은 해당 질의 노드에 대한 XML 엘리먼트가 문서 D 상에 존재하지 않는다는 의미이다. 이 경우, 가지 패턴 전체에 대한 해는 D에 존재하지 않으므로, 가지 패턴 질의 처리를 완전히 끝내지 않더라도, 해당 가지 패턴의 해가 D에 있는지 검사할 수 있는 방법이 된다.

bitPath 기법에서는 비트맵 인덱스의 특성을 이용하여, 인덱스만으로 OR, NOT 을 포함하는 가지 패턴을 효과적으로 처리할 수 있다. 그림 8은 그림 2(d), (e)에서 보인 OR와 NOT을 포함하는 각 가지 패턴에 대해 어떻게 비트벡터를 구성하여 질의 노드에 할당하는지를 보인다.

그림 2(d)와 같은 질의는 XPath 표현식 //A/B[C or D]//E로, 그림 2(e)와 같은 질의는 //A/B[not(./C)]//E 로 표시된다. 먼저 OR의 경우에는 OR로 연결된 두 질의 노드를 대표할 새로운 질의 노드 X를 생성하고, 여기에 원 질의 노드 C와 D에 대한 비트벡터들을 비트열-OR로 병합하여 할당함으로써 간단하게 제공할 수 있다. NOT의 경우에는 NOT선택 조건을 가진 질의 노드를 q라 할 때 그 부모나 조상 노드는 이 q를 포함하지

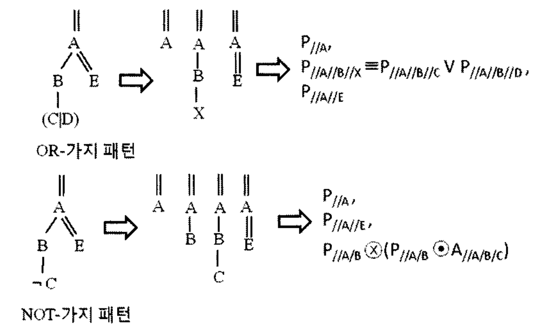


그림 8 OR, NOT을 포함한 가지 패턴의 처리

표 1 질의 처리 기법의 요약

기 법	bitTag	bitPath	bitTag⁺	descTag	bitTwig
이용 인덱스	*bitTag*	*bitPath*	*bitTag*⁺	*bitDesc* ∪ *bitTag*	*bitPath* ∪ ***bitAnc***
질의 노드 당 입력	단일 비트벡터와 레이블 값	단일 비트벡터와 레이블 값	여러 비트벡터와 레이블 값	단일 비트벡터와 레이블 값	여러 비트벡터, 레이블 이용안함
비트벡터의 유형	bitTag(a)	$\bigvee_{i \in leaf(a)} bitPath(i)$	$bitTag^{+}(a, l)$	$\left(\bigvee_{i \in pids(s)} bitDesc(i)\right) \odot bitTag(a)$	$\bigcup_{i \in pid(s)} \binom{bitAnc(i),}{bitPath(i)}$
최적화	advance(k)	condense(q)	advance(q, k)	advance(q, k)	condense(q)

않아야 조건이 성립된다. 먼저 NOT 선택 조건을 가진 질의 노드 q에 대한 *bitAnc*의 비트벡터 A를 구한 후, 이를 ancestor(q) 또는 parent(q)의 *bitPath*의 비트벡터 P와 비트열-AND 연산을 수행하여 NOT 선택 조건을 가진 질의 노드를 자식 또는 후손으로 가지는 ancestor(q) 또는 parent(q)에 대한 비트벡터를 구한다. 그리고 이를 ancestor(q) 또는 parent(q) 자신의 비트벡터와 비트열-XOR연산을 수행함으로써 질의 노드 q를 포함하지 않는 ancestor(q) 또는 parent(q)의 비트벡터를 구한다.

5.3 커서 이동

홀리스틱 가지 패턴 조인을 위해, 우리는 비트맵 인덱스에 커서 기능을 부여하였다. 그래서, 한 비트벡터에서 1의 위치(ROWID)는 커서를 전진시키면서 차례로 획득할 수 있다. 함수 advance(q)는 질의 노드 q의 비트벡터의 커서를 현재 커서 위치에서 가장 가까운 다음의 1로 이동시키고, q에 할당된 입력 엘리먼트 정보를 담는 변수 $Curr_q$를 1의 위치 값이나 또는 레이블 값으로 대체한다. $Curr_q$는 bitTwig기법에 대해서는 (*pos*, *pid*, *levelNum*)의 값을 다른 기법에 대해서는 pos-번째 열에서 레이블 값(*startPos*, *endPos*, *levelNum*)을 읽어 적재한다.

질의 노드에 할당된 비트벡터가 단일인 경우, $Curr_q$는 항상 커서가 가리키는 1의 위치, pos를 가진다. 질의 노드에 할당된 비트벡터가 복수개인 경우에는 각 비트벡터에서의 커서가 가리키고 있는 pos 값 중 가장 작은 값이 $Curr_q$로 선택된다. 이는 알고리즘 4에서 각 질의 노드에서 한번에 하나의 엘리먼트만이 접근되게 한다. 그림 9는 bitPath기법에서 그림 2(a)의 가지 패턴에서 질의 노드 A에 할당되는 두 비트벡터와 이때 질의 노드의 현재 엘리먼트 값 $Curr_q$의 상태를 보인다.

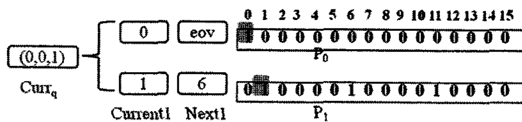


그림 9 복수 개의 비트벡터와 커서 값

그림에서 Current1과 Next1은 각 비트벡터의 현재 커서의 위치와 미리 보기를 통한 다음 1의 위치를 저장한다. Next1를 통하여 우리는 알고리즘 3에서 A-D 관계성 검사를 위해 비트벡터에서 임의 구간을 검색하는 것을 피할 수 있다. Current1 중에서 가장 작은 값이 $Curr_q$으로 선택되며, 이때 해당 비트벡터가 대표하는 *pid*와 *levelNum*도 같이 복사한다.

advance()가 호출될 때, 현재 $Curr_q$를 구성한 Current1의 값은 그 자신의 Next1으로 대체되며, 비트벡터의 커서는 Next1 다음의 1을 찾아 전진하고, 새 커서의 위치가 Next1으로 설정된다. 그림 9에서 현재 두 비트벡터의 Current1은 각각 0, 1을 가리키며, 따라서 가장 작은 위치 값인 0과 P_0의 pid 0 그리고 이의 *levelNum* 값을 $Curr_q$에 설정한다. 다음으로 advance(q)가 호출이 되면, 먼저 현재 $Curr_q$를 구성한 P_0의 Next1을 Current1으로 옮기고, 커서를 이동하는데, EOV이므로 더 이상 진행하지 않는다. 그리고, 현재의 두 Current1, EOV와 1 중 작은 값 1과 대응하는 *pid*, *levelNum*을 $Curr_q$로 설정한다(1, 1, 2).

bitTag$^+$ 기법에서는 또한 실행 중에 parent(q) 또는 ancestor(q)의 입력의 레벨 값에 따라 비트벡터의 current1을 $Curr_q$로 옮기는 과정을 제한함으로써 I/O 최적도를 증가시킬 수 있다. 예를 들어, 여러 개의 비트벡터가 할당된ancestor(q)가 있다고 하고, $Curr_{ancestor(q)}$.level이 이 ancestor(q)의 현재 입력의 레벨 값이라 하자. 이때 우리는 levelNum > $Curr_{ancestor(q)}$. level 조건을 만족하지 않는 비트벡터에서 Current1을 $Curr_q$로 복사하는 것을 막는다. P-C 관계성을 가지는 두 질의 노드에 대해서는 levelNum = $Curr_{parent(q)}$.level+1 조건을 만족하지 않는 비트벡터에서 Current1을 $Curr_q$로 복사하는 것을 막는다. 이렇게 함으로써 우리는 불필요한 엘리먼트에 대한 레이블 값을 테이블로부터 읽는 것을 피하고, 이를 통해 I/O 최적도를 향상시킨다(이 기법은 *iTwigJoin*[14]에서 I/O 최적도의 향상을 위한 물리적 데이터 분할 기법인 tag+Level 로 소개되었다.)

5.4 성능 최적화

한 질의 노드에 대해 복수 개의 입력을 허용하는 홀리스틱 가지 패턴 조인 알고리즘에서의 CPU 시간은 가지 패턴 Q에 대해 O(N x |Q| x |Input + Output|)으로 알려져 있다[14]. 여기에서 N은 입력 중 가지 패턴 Q에서 유용한 입력 리스트의 수이며, |Q|는 질의 노드의 수, |Input|과 |Output|은 각각 입력/출력 데이터의 크기이다. 여기에서 |output|은 모든 알고리즘에서 공통된 결과를 제공해야 하므로, 언제나 일정하고, 따라서 성능 향상을 위해서는 |Q|나 |Input|을 줄일 수 있는 방법이 고안될 수 있다. 이 절에서는 이들을 줄임으로써 가지 패턴 조인의 성능을 향상시키기 위한 기법을 소개한다.

5.4.1 질의 노드의 제거

선형 경로를 값 도메인으로 한 비트맵 인덱스를 이용하는 경우, 즉 bitPath와 bitTwig에서는 가지 패턴의 질의 노드 중 단말 노드나 브랜칭 노드(두 선형 경로의 접점)가 아니면서, 결과로서 선택되지 않는 질의 노드들은 가지 패턴 질의 처리 시 배제할 수 있다. 예를 들어, 그림 2(c)의 가지 패턴은 XPath 표현식으로 //A[B/C]//D로 표현되는데 이 때 B질의 노드는 가지 패턴 조인 시

배제한다. 이러한 질의 노드의 삭제는 condense(q)를 통해 수행된다. 이러한 질의 노드의 배제는 |Q|와 |INPUT| 모두를 줄이면서, 가지 패턴 결과의 정확성은 그대로 유지한다. 왜냐하면 이 예에서, 질의 노드 C의 입력들은 정리 1.2에 따라 이미 경로 //A/B/C의 인스턴스이기 때문에 따로 B엘리먼트들이 B와 C간의 P-C 관계성 검사를 위해 필요하지 않기 때문이다. 여기에서 A와 C 간에 A-D 관계성이라면 레벨 비교가 필요하지 않지만, 이 예에서와 같이 P-C와 연결이 된 경우에는 A와 C의 레벨차이가 2이므로 질의 노드 B는 제외하더라도 가지 패턴 조인 시 A와 C간의 레벨 값 차이의 비교는 수행해야 한다.

5.4.2 불필요한 레이블 값 읽기의 건너뜀

Bruno *et al.* [12]은 XB-tree라 명명된 인덱스화된 입력 리스트를 이용하여 가지 패턴 조인 시 전체 데이터를 다 읽지 않고 불필요한 엘리먼트에 대한 레이블 값을 읽는 과정을 건너뛰는 방법을 제안하였다. 이 방법의 기본 아이디어는 그림 10과 같다. 다음과 같은 경로 질의 //A//B//C가 있다고 하고, 각 입력 리스트에서 엘리먼트들의 구간 값은 그림 10과 같다. 그리고, 각 입력 리스트의 커서는 현재 a_i, b_{k-1}, c_{k-1}을 가리킨다고 하자. 건너뜀 이 없으면, $list_B$의 엘리먼트 b_{k-1}부터 b_i까지 모두 읽힐 것이다. 여기에서 b_i 엘리먼트 만이 a_i와 A-D 관계성 조건 startPos(A) < startPos(B) ∧ endPos(B) < endPos(B) 조건을 만족한다. 따라서, $List_B$의 B_k에서 b_{i-1}까지의 레이블 값들은 단순히 startPos(a_i) < startPos(b_i)인 b_i 를 찾음으로써 읽어 들이는 것을 피할 수 있다. 이 과정은 재귀적으로 처리하여 $List_C$에서도 C_k 부터 c_{i-1}까지 노드를 건너뛸 수 있다.

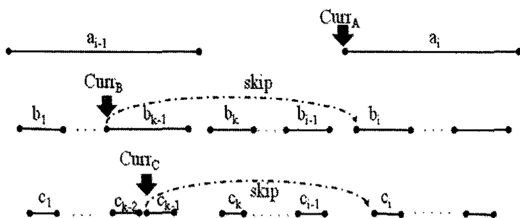


그림 10 불필요한 레이블 값의 건너뜀

이 논문에서는 이러한 불필요한 레이블 값 읽기를 알고리즘 5에서 보이는 advance(q,k) 라는 간단한 함수를 통해 지원한다. 여기에서 q는 커서 이동을 수행할 질의 노드 이며, k는 pos(ancestor(q)), 즉 조상 노드에 대응하는 비트벡터의 현재 커서 위치이다. 두 노드 x, y가 A-D 관계성을 가지고 있다면, 정리 2.1에 따라 pos(x) < pos(y)의 조건을 우선 만족한다. 알고리즘에서는 후

```
   입력: 질의 노드 q, 조상 노드에 대응하는 1 의 위치
1  k
2  pos ← C.position; //커서 C 의 현재 논리적 위치
3  if k 값이 주어지지 않음 then
       pos ← getPosOfNext1(v); //v 는 현재 질의 노드
4  q에 대응하는 비트 벡터
5  else
6      repeat
7          pos ← getPosOfNext1(v);
8      until pos ≤ k;
9  end
10 if pos ≠ EOV then //eov : end of bitvector
       node 테이블에서 pos 가 가리키는 열에서 레이블
11 값을 읽음;
   end
```

알고리즘 5 단일 비트벡터에서의 advance(q, k)

손 질의 노드의 커서를 이 조건을 만족할 때까지 반복적으로 전진시킴으로써(5-7)로써 불필요한 엘리먼트에 대한 레이블 값을 읽기 위해 테이블에 접근하는 것을 피한다. 여기에서 알고리즘 5의 advance(q, k)는 질의 노드 당 단일 비트벡터일 때의 버전이다. 복수개의 비트벡터에 대한 advance(q, k)는 q에 할당된 모든 비트벡터의 커서를 k위치까지 전진시키고, 5.3 절의 설명에 따라 $Curr_q$ 값을 설정한다.

선형 경로 도메인에서의 인덱스를 이용하는 경우 advance(q, k)를 이용한 입력 데이터 읽기의 건너뜀을 필요로 하지 않는다. 이는 아래에 모순증명법으로 증명한다.

정리 2. 경로 단위로 분할된 입력을 가지고, 임의의 선형 경로를 만족하는 모든 후손 노드들은 해당 선형 경로의 해에 유용하며, 가지 패턴 조인 과정 중에 건너뛰기 될 수 없다.

보조정리 2. a_p와 d_p를 한 선형 경로 p, //A//D를 만족시키는 $a_p \in list_A$, $d_p \in list_D$ 의 각 엘리먼트라 한다. 그러면, $\forall d_p$ 는 언제나 $\exists a_p$의 후손 엘리먼트이다.

증명. 선형 경로 p를 만족하면서 건너뛰기 될 수 있는 후손 엘리먼트 d_p가 있다고 가정하자. 이는 어떠한 i에 대해, d_p가 $a_{p_{i-1}}$과 a_{p_i}사이에 존재한다는 의미이다. 이를 위해서는 d_p는 $\exists a_p \in list_A$의 후손 엘리먼트이면 안 된다. 이것은 정리 3의 $\forall d_p$가 $\exists a_p$의 후손 엘리먼트라는 가정을 위배한다. □

5.5 종합과 분석

그림 11은 그림 2(c)의 가지 패턴에 대해 그간 설명된 알고리즘들을 가지고 질의 처리 하는 예를 보인다. 여기에서는 *bitPath*를 이용한다고 가정한다. 먼저 condense(q)를 통해 질의 노드 B를 제거한다. 이를 통해 B 노드에 대한 입력과 스택을 모두 제거한다. 다음으로, 각 질의 노드에 각 질의 노드의 경로에 대응하는 비트

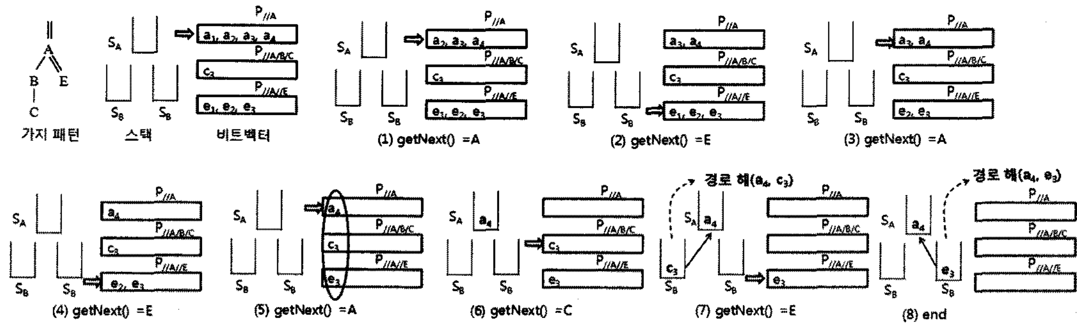


그림 11 홀리스틱 가지 조인의 예

벡터들을 할당한다. (편의를 위해 실제 엘리먼트 이름을 기재하였으나, 실제 입력은 비트벡터이다.) 초기에는 루트 질의 노드인 q에 대한 비트벡터 $P_{//A}$를 가리킨다. 각 질의 노드의 $Curr_q$를 통해 이들 간에 A와 E간에 A-D 관계성이 있는지, A와 C 간에 A-D 관계성이 있으면서 레벨 차이가 2인지를 비교하여 관계성이 없으므로 아무 작업 없이 getNext()를 호출한다. (이때 관계성 비교는 실제 테이블로부터 가져온 레이블 값이나, bitTwig인 경우 checkAD()를 이용해 수행한다.) (1)-(4)까지는 질의 노드들의 $Curr_q$들이 서로 관계성을 만족시키지 못하므로 아무런 작업도 진행되지 않고, getNext()로 가장 작은 pos값을 가지는 비트벡터들만 전진시킨다. (5)에서는 a_4가 다른 질의 노드의 현재 엘리먼트와의 관계성을 만족시키므로 스택에 삽입하고, (6)에서는 조상 노드의 스택이 채워져 있으므로 c_3가 스택에 삽입되었다가 단말 노드라는 조건에 의해 경로 해 (a_4, c_3)가 출력되고 c_3는 스택에서 삭제된다. 다음으로 e_3 또한 조상 노드의 스택에 a_4 와 연결되어 경로 해 (a_4, e_3)를 출력하고, 삭제된다. 그리고, 이 두 경로해는 mergeAllPathSolutions()으로 정렬-합병되어 최종적으로 가지 패턴의 해 (a_4, c_3, e_3)를 출력한다.

이 논문의 가지 패턴 질의 처리에서 3.1절의 기본 인덱스를 이용하는 경우, 입력 데이터의 크기는 $O(\sum_i^n|v_i|+|Records|)$이다. 여기에서 $|v_i|$는 한 비트벡터의 크기이며, n은 질의 처리 과정 중에 이용된 비트벡터의 수, |Record|는 접근된 레코드의 크기이다. 따라서, 5.4절에 따라 이 때의 CPU 시간은 $O(N \times |Q| \times \sum_i^n|v_i|+|Records|$ 이 된다. 이는 홀리스틱 가지 조인의 CPU 시간에 인덱스로부터 질의 처리에 유용한 비트벡터들을 메모리에 적재하는 시간이 포함된 것이다.

반면에 bitTwig 기법의 경우, 두 인덱스 만을 필요로 하므로, 입력 데이터의 크기는 $O(2 \times \sum_i^n|v_i|)$이다. 이의 CPU 시간은 알고리즘 4의 각 부함수의 CPU 시간과 이들이 전체 알고리즘에서 호출된 횟수를 통해 계산한다. 알고리즘 4는 부함수 advance(), getNext(), cleanStack()을 포함한다. advance()는 알고리즘 4에서 유용한 비트벡터들의 모든 1들을 한번 읽는다. 그리고, 각 advance() 호출마다, 비트벡터의 current1들 중 가장 작은 pos 값이 선택된다. 그러므로, advance()의 알고리즘에서의 전체 비용은 $O(N \times |Q| \times \sum_i^n|P_i|$이다. 알고리즘 3의 checkAD() 는 cleanStack()과 getNext()에서 A-D 관계성 식별을 위해 이용된다. checkAD()는 *bitAnc* 의 비트벡터A_i만의 커서를 특정 위치로 이동시키므로, checkAD()는 advance() 호출 비용에 영향을 미치지 않는다. getNext() 는 checkAD()를 조상 질의 노드의 커서 위치 값을 가지고 호출하며, 이는 언제나 advance()를 통해 순방향으로 전진된다. 따라서 비록 checkAD()가 Ai의 커서를 특정 위치로 이동시킨다 하더라도, 그 위치는 한 비트벡터에서는 언제나 순방향으로 이동되는 특성을 가지므로, getNext()의 전체 호출 비용은 $O(N \times |Q| \times \sum_i^n|A_i|$이다. 반면에, cleanStack($x_i$, y_j) 호출은 A-D 관계성 판별을 위해 스택 S_x의 항목을 비트벡터 A_j의 현재 커서 위치와 비교한다. 따라서, S_x의 항목들은 스택의 맨 위부터 맨 아래까지 순서대로 방문되며, 이는 정리 2.2에 따라 A_j의 커서를 역방향으로 움직이게 한다. 하지만, 이런 경우에서도 커서는 한번 역방향으로 방문한 1들을 넘어 반복적으로 역방향으로 이동하지는 않는다. 왜냐하면, 가장 위의 스택 항목이 x_j와 A-D 관계성을 가지는 경우 그 밑의 다른 스택 항목들은 더 이상 x_j와 비교되지 않으며 이는 역방향 커서 이동을 더 이상 유발하지 않는다. 반대로, 맨 위의 스택 항목이 x_j와 A-D 관계성을 가지지 않는 경우, 해당 항목은 S_x로부터 빠지며(pop), 이 항목이 표시하였던 위치 값으로의 커서 이동은 다시 발생하지 않는다. 따라서, 이 논문에서 단방향 이동만 가능한 2개의 커서를 가지고, cleanStack() 호출의 알고리즘에서의 전체 비용은 $O(N \times |Q| \times \sum_i^n|A_i|$이다. 최종적으로, 경로 해의 작성을 포함한

bitTwig기법의 전체 CPU 시간은 $O(N \times |Q| \times (\sum_{i}^{n}(|A_i| + |P_i|) + |Output|))$이다. 여기에서 $\sum_{i}^{n}(|A_i| + |P_i|)$는 대부분의 경우 구간-기반 레이블링 기법에 따른 레이블 값보다도 작은 크기이다(표 4 참조).

6. 실험 및 성능 평가

6.1 실험 환경

이 논문에서의 알고리즘들은 GNU C++ 버전 4.3으로 단일 스레드 실행 코드로 구현되었다. 모든 실험은 Intel Core2 Duo 2Ghz와 5,400 RPM의 2.5' HDD, Fedora 리눅스 10이 설치된 시스템에서 수행되었다. 우리는 또한 비트맵 인덱스의 구현을 위해 FastBit V.0.9.8[23]의 비트벡터에 커서 기능을 삽입하여 이용하였으며, XML 엘리먼트의 파싱을 위해 Xerces C++ XML 라이브러리를 이용하였다. 비트맵 인덱스들과 그 이용 기법이 얼마나 홀리스틱 가지 패턴 조인의 성능을 개선하는지를 비교하기 위해 *twigStack*[12]과 우리의 알고리즘을 비교하였다.

실험을 위한 데이터로는 1 개의 합성 XML과 3 개의 실제 XML 응용을 이용하였다. 모든 데이터들은 먼저 파싱을 통해 구간 기반-레이블 값을 가지고 그림 3(b)의 테이블에 튜플들로 저장된다. XMark는 프로그램을 통해 생성되는 합성 데이터로 가상의 경매 사이트에서 다루어지는 데이터를 XML로 표현한다. 이 XMark의 구조는 다소 재귀적(recursive)이나, 생성된 XML 트리의 깊이(depth)는 크게 깊지 않다. DBLP와 SWISS-PROT(줄여서 SPORT)는 논문의 서지와 단백질 정보의 표현을 위해 작성되는 실제 XML 데이터로 DBLP에 비해 XML 트리의 깊이가 깊지 않다. TREEBANK(줄여서 TBANK) 또한 언어의 구조적 표현을 위해 사용되는 실제 XML 데이터로 위의 세 데이터와 비교해 트리의 깊이가 깊고, 보다 재귀적인 특성을 가진다. 이들 데이터들의 통계 정보는 표 2에서 보인다. 보면 XML 데이터에서 유일한 태그 이름의 수는 큰 차이가 없는 반면, 서로 유일한 선형 경로의 수에는 큰 차이가 존재한다(표 2의 TBANK).

표 2 XML 데이터의 통계

	XMark	DBLP	TBANK	SPROT
데이터 크기(MB)	113	486	86	112
엘리먼트 수	1,666,315	11,692,273	2,437,666	2,977,031
애트리뷰트 수	381,878	2,305,980	1	2,189,859
태그 수	77	41	251	99
태그+레벨 수	119	47	2,237	100
선형 경로 수	548	170	338,749	264
최대/평균 깊이	12/5	6/2.9	36/7.8	5/3.5

표 3 가지 패턴 질의와 그 결과의 수

이름	질 의	결과 수
XMark1	//site//closed_auction[.//buyer][.//price]//date	9,750
XMark2	//people//person[.//address/zipcode]/profile/education	3,241
XMark3	//item[./location]/description//keyword	26,946
XMark4	//site/regions[.//asia]//listitem/text	25,425
DBLP1	//inproceedings//title[.//i]//sup	324
DBLP2	//inproceedings[title][editor]/publisher	23,059
DBLP3	//inproceedings[author][title]//booktitle	7
DBLP4	//phdthesis[.//author][school][title]/url	2
TBANK1	//S/VP[IN]//NP	20,311
TBANK2	//S/VP/PP[IN]/NP/VBN	152
TBANK3	//VP[DT]//PRP_DOLLAR_	3
TBANK4	//S//NP[.//PP/TO][VP/_NONE_]//JJ	2
SPROT1	//Entry[.//Org][.//ZN_FING]//@from	18,562
SPROT2	//Entry/Ref[DB]/Cite	76,685
SPROT3	//Entry/Features[.//SITE]//DOMAIN/Descr	9,411

실험에 사용된 가지 질의 패턴들은 표 3에서 보인다. 이 가지 패턴들은 몇몇 홀리스틱 가지 패턴 조인 기법들에서 I/O 최적(optimal)하다고 알려진 가지 패턴 클래스와 그 외의 클래스들을 포함한다. 즉 (1) A-D 축으로만 구성된 가지 패턴 (2) P-C축으로만 구성된 가지 패턴 (3) 접점(branching node)이 그 자식 노드와 P-C 관계성을 가지지 않는 가지 패턴 (4) (1)-(3)을 제외한 A-D, P-C 관계성 모드를 포함하는 가지 패턴으로 구성된다.

6.2 XML 데이터의 저장과 인덱스의 구축

이 논문에서의 비트맵 인덱스들은 그림 3(b)의 node 테이블을 기준으로 작성된다. XML 데이터를 저장할 때 우리는 역 경로 문자열(reverse path string)과 구간-기반 레이블 값을 작성하기 위해 하나의 스택을 이용하여 저장한다. 그리고 매 이벤트마다 값을 1씩 증가시키는 계수기를 둔다. 인덱스 구축 알고리즘은 *startElement* 이벤트 발생 시 엘리먼트를 스택에 삽입(push())하고, *endElement* 이벤트 발생 시 스택에서 엘리먼트를 꺼낸다(pop()). push() 호출 시 계수기 값을 *startPos*로 부여하고, 또한 해당 엘리먼트의 역 경로 문자열은 가장 최상위부터 바닥까지 스택에 쌓인 엘리먼트들의 태그 이름을 서로 연결시킴으로써 구한다. 그리고 유일한 경로 ID를 이 엘리먼트에 부여한다. 그리고, pop() 시에는 *endPos*를 부여한다. *levelNum*은 스택에 쌓인 엘리먼트의 수로 계산된다.

비트맵 인덱스의 각 비트벡터는 단일 또는 두 개의 컬럼에서 하나의 구별되는 유일한 값을 대표한다. 따라서, 3.1절의 기본 인덱스들의 비트벡터의 생성은 RDBMS에서 일반 비트맵 인덱스를 생성하는 방법과 다르지 않다. 하지만 3.2절의 *bitAnc*와 *bitDesc*는 각 비트벡터가 하

나의 구별되는 유일한 값이 아닌 여러 값들, 즉 조상 노드들과 후손 노드들을 가리키므로, 다른 방법으로 생성된다. *bitAnc*를 구축하기 위해, push()가 호출될 때마다, 스택에 쌓여있는 모든 엘리먼트들에 새로 삽입되는 엘리먼트의 pid들을 복사한다. 이렇게 되면, 파싱이 끝났을 때, 모든 엘리먼트는 그들의 모든 후손 노드들에 대한 pid 값들을 가지게 된다. 예로, 그림 1의 루트 a1은 모든 엘리먼트들의 pid 0~11을 가진다. 그리고 나서 i-번째 엘리먼트가 가지고 있는 pid집합에 대해, 각 pid들에 대응하는 비트벡터들의 i번째 비트를 1로 설정함으로써 *bitAnc*를 구축한다. *bitDesc*를 구축하기 위해서는 pop()이 호출될 때마다 현재 스택에 쌓여있는 엘리먼트들의 pid들을 꺼내지는 엘리먼트에 복사한다. 그러면, 모든 엘리먼트는 그들의 조상 엘리먼트들이 가지는 pid들의 집합을 가진다. 예로, 그림 1의 b1은 pid 0, 1,2를 가진다. 그리고 나서 이들 각 pid들에 대해 비트벡터를 구축하여 *bitDesc*를 구축한다.

6.3 실험 결과

6.3.1 실험의 척도

이 실험에서 비교 척도로는 인덱스의 크기(표 4), 질의 처리 실행 시간(그림 12), 그리고, 질의 처리 동안에 읽어지는 데이터의 크기(그림 13)를 이용한다. 이를 통해 이 논문에서의 인덱스와 기법들이 원 데이터보다 얼마나 작으면서, 질의 처리 시 얼마나 데이터를 보다 적게 필요로 하고, 기존 홀리스틱 가지 조인 기법보다 얼마나 효율적인지를 보인다.

그림 12에서는 이 논문에서의 6개의 인덱스 기법과 twigStack을 4개의 XML 데이터와 15개의 질의를 가지고 비교한다. 그림에서 bitTag는 *bitTag* 인덱스를 이용한 가지 패턴 조인 기법을, tagSkip은 *bitTag* 인덱스를 이용하면서 advance(q, k)를 이용한 불필요한 데이터 의 건너뛰기 하는 인덱스 기법을 의미한다. 다른 기법들은 모두 advance(q, k)나 condense(q)를 이용해 최적화 되었다. 실험에서 인덱스는 메모리 상에 현재 적재되지 않았으며, 버퍼 크기 또한 0으로 설정되었다. 따라서, 이 실험 결과는 실제 질의 처리 시간에 인덱스를 디스크로부터 메모리에 적재되는 시간까지 포함한다.

6.3.2 인덱스 통계

인덱스 크기와 구축 시간의 비교는 표 4와 표 5에서 보인다. 각 인덱스의 크기는 인덱스에서WAH로 압축된 비트벡터들의 총 크기를 합한 값이다. 인덱스에서 비트벡터의 수는 선정한 도메인의 수에 따른다. 예로, XMark에 대한 태그 단위의 비트맵 인덱스 *bitTag*는 XMark의 태그 수, 77개의 비트벡터를 갖는다. 원 데이터뿐만 아니라 구간-기반 레이블 값과 비교해서도 이 논문의 인덱스들의 크기는 충분히 작다. 또한 인덱스 2개를 이

표 4 인덱스의 크기

단위(MB)	XMark (113MB)	DBLP (486MB)	TBANK (86MB)	SPROT (112MB)
구간 기반 레이블	23.44	160.22	27.90	59.13
bitTag	5.32	19.67	6.85	14.40
bitPath	6.18	20.00	23.53	19.03
bitAnc	6.74	20.08	30.84	26.50
bitDesc	6.00	18.34	24.18	18.92
bitTag+	6.05	20.67	14.33	14.62

표 5 인덱스 생성 시간

단위(초)	XMark	DBLP	TBANK	SPROT
bitTag	17.36	98.51	38.61	52.34
bitPath	20.81	77.63	7,408.09	35.71
bitAnc	34.45	169.10	8,054.15	69.11
bitDesc	21.73	86.35	7,528.23	38.28

용하는 bitTwig의 경우, *bitPath*와 *bitAnc* 인덱스의 합 또한 TBANK을 제외한 나머지 모든 경우에서 구간-기반 레이블 값보다도 작음을 확인할 수 있다. 여기에서 관찰된 내용 중 하나는 XML 문서 구조의 깊이가 깊고, 재귀적이지 않는 한 선형 경로를 값-도메인으로 구축된 비트벡터들의 크기는 충분히 작다는 것이다.

관찰된 또 다른 내용은 인덱스 구축 시간은 릴레이션의 튜플 수보다는 애트리뷰트의 카디널리티에 주로 영향을 받는다는 것이다. 예를 들어 TBANK에서 선형 경로를 값-도메인으로 인덱스를 구축하는 경우 많은 시간이 소요되었다. 반면에 1400만개 이상의 엘리먼트, 즉 튜플을 가지는 DBLP에 대해 인덱스를 구축하는 것은상대적으로 작은 시간이 소요되었다.

6.3.3 성능 분석

질의 처리 과정 중 읽어 들인 데이터 크기 면에서(그림 12 참조) 보면, bitTwig 기법이 모든 경우에 대해 질의 처리 시 가장 적은 양의 데이터를 읽는다. 이는 bitTwig 에서는 가지 패턴 질의의 해를 구하기 위해 두 비트맵 인덱스로부터의 비트벡터들만을 필요로 하기 때문이다. 또한 선형 경로를 값-도메인으로 하여 구축된 인덱스를 이용할 시 보다 적은 크기의 레코드를 필요로 함을 확인할 수 있다. 일반적으로, 태그 단위로 비트벡터를 구축할 시 가장 많은 레코드를 읽으며, advance(q, k)가 이 레코드 크기를 크게 줄임을 확인할 수 있다. (tagName, level) 도메인 상에 구축된 *bitTag+* 인덱스를 이용하는 tagPlus 기법은 읽어지는 인덱스의 크기와 데이터 레코드의 크기 간의 trade-off가 있다. 이 trade-off는 특히 TBANK 데이터에서 두드러지게 나타난다. tagPlus 기법은 bitTag와 비교해 상대적으로 적은 데이터 레코드를 읽지만 대신에 레코드 선정을 위한

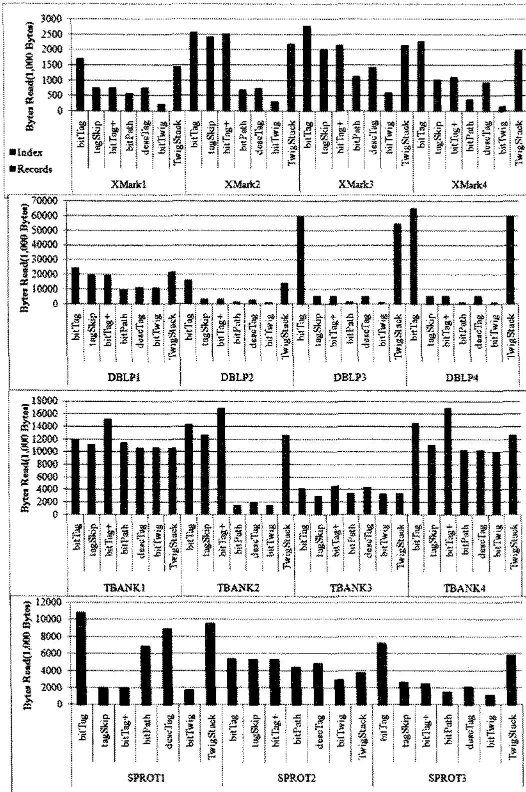


그림 12 질의 처리 과정 중 읽어들인 데이터 크기

보다 많은 비트 벡터를 필요로 한다. bitTag+는 또한, 질의 처리 시간에서도 다른 인덱스 기법보다 향상된 결과를 보여주지는 못하였다. 임의 경로의 인스턴스의 단말을 루트로 하는 부분 트리에서 해당 부분트리에 속하는 노드들을 대표하는*bitDesc*의 비트벡터와 *bitTag*의 비트벡터를 비트열-AND로 병합하는 bitDesc 기법은 선형 경로 기반의 다른 인덱스 기법과 비교하여 나은 성능을 보이기도 하나, XML 구조가 단순하고, 단말 노드의 숫자가 많은 경우, 비트벡터의 병합 비용이 성능에 영향을 미치기도 한다. 예. 그림 13의 SPROT1.

질의 성능 면에서 보면, bitTwig는 대부분의 경우 다른 인덱스 기법들을 압도한다. 특히, DBLP나 SPROT와같이 XML 데이터의 깊이가 깊지 않고, 재귀적이지 않은 경우에는 다른 기법들을 성능 면에서 매우 큰 차이로 압도하였다. 하지만, 깊이가 깊고, 재귀적인 TBANK 데이터에 대해서는 가장 나쁜 성능을 보여 주었다. 이는 두 가지 문제에 기인한다. 첫째로는 재귀적인 XML 문서에서는 경로 수가 많아 주어진 가지 패턴에 대해 그만큼 찾을 경로의 수와 대응하는 비트벡터의 수가 많아진다. 따라서, 그만큼 유효한 비트벡터를 찾는 비용이 높아진다. 둘째로, 이 경우 한 질의 노드에 대해 많은 비트벡터들을 유지하고 커서 이동을 관리해야 하므로, 이에 대한 비용이 증가한다. 이러한 XML 데이터에 대해서는 tagSkip 기법이 가장 최적의 질의 성능을 보여주었다. tagPlus는 어떠한 경우에도 최적이지는 않았지만, 마찬가지로 TBANK 데이터에 대해서는 tagSkip과 bitTag 다음으로 빠른 성능을 보여주었다. bitDesc는 bitPath와 유사한 질의처리 성능을 보였지만, TBANK에 대해서는 bitPath보다 나았다.

우리는 또한, TBANK 데이터에 대한 bitPath와 bitDesc 기법에서 대부분의 실행 시간이 주어진 가지 패턴에 대하여 유효한 비트벡터를 찾고, 이를 디스크로부터 메모리에 적재하는데 소요됨을 파악하였다. //을 포함하는 경로 표

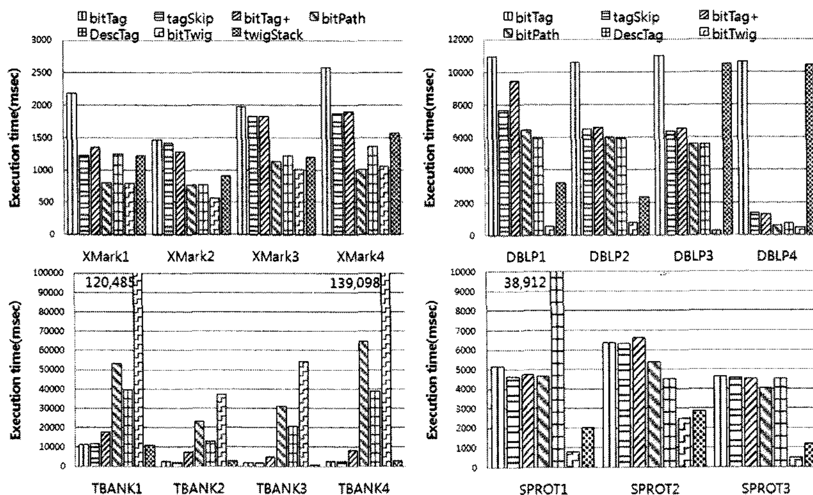


그림 13 질의 실행 시간의 비교

현식을 가지고 유효한 비트벡터들을 찾는 일은 효과적으로 지원하기 어렵다고 알려진 서브시퀀스 매칭 (subsequence matching) 문제 이다. 이러한 경우 **미리 계산된 조인**(pre-computed join) 기법과 유사하게, 한번 검색된 경로문자열에 대하여 유효한 경로 기반의 비트벡터들을 나중을 위해 비트열-OR로 병합해 따로 저장하는 방법이 제시될 수 있다. 일단 비트벡터들을 주어진 경로 문자열에 대하여 병합하는 경우, 이후에는 주어진 경로 문자열에 대해서 일치 검색(exact matching)으로 하나의 비트벡터를 찾아 적재하면 되므로, 서브시퀀스 매칭 검색을 피하면서, 적재할 비트벡터의 수도 줄이는 효과가 있다.

7. 관련 연구

Bruno 등의 *twigStack*[12]은 가지 패턴 매칭 과정 시 불필요한 경로 해 작성을 피하도록 고안된 다방향 정렬-합병 조인(multi-way sort-merge) 조인이다. 이 기법은 // 축으로만 구성된 가지 패턴에 대해 I/O 최적이라고 증명된 바 있다[20]. twigStack 이후로 많은 연구들은 XPath의 더 큰 클래스에서 I/O 최적도를 보장하는 방법을 구하고자 수행되었다. *twigStackList*[16]는 미리 보기 리스트(look-ahead list)를 따로 메모리에 두어, I/O 최적도가 보장되는 가지 패턴 클래스를 // 뿐만 아니라, 접점과 그의 자식 노드들이 P-C로 연결되지 않는 가지 패턴으로까지 확장하였다. *iTwigJoin*[14] 에서는 태그 단위가 아닌 (태그 이름, 레벨) 단위나 또는 선형 경로 단위로 XML 데이터를 디스크에 물리적으로 분할 저장하고, 이를 가지고 가지 패턴 조인을 수행하였을 때, I/O 최적도가 /-축만을 가진 가지 패턴이나 또는 하나의 접점만을 가지는 가지 패턴에 대해 보장됨을 보였다. Bruno 등은 또한 입력 리스트를 중간 노드들이 하위 노드들을 포함하는 구간 값을 가지도록 변형된 B+트리 인덱스화하는 방법을 제안하였다. 이는 홀리스틱 가지 조인 알고리즘에서 불필요한 입력 데이터를 건너뛰게 만들어 읽어 들일 데이터의 양을 줄이게 하였다. *XR-tree*[13] 또한 스태빙 리스트(stabbing list)라는 별도의 리스트를 중간 노드들이 갖는 B+트리의 변형으로 이러한 건너뛰기를 구현하였다. Kaushik 등[19]은 태그 단위로 분할된 입력 리스트의 각 엘리먼트를 DataGuide와 같은 인덱스 그래프(index graph)와 연결시킴으로써 건너뛰기를 하는 방법을 구현하였다. *TJFast*는 XML 데이터의 레이블링 기법을 각 엘리먼트의 경로 문자열을 인코딩시킬 수 있도록 변형 듀이 레이블링 기법으로 대체하고, 이를 이용하여 가지 패턴의 단말 노드에 대해 입력 리스트와 스택을 유지하게 함으로써 입력 데이터의 양을 줄이는 방법을 제시하였다. OR, NOT 연산자를 가지는 가지 패턴의 지원에 대해서는 [17,18]에서 수행되었다.

XML 데이터를 관계형 DBMS에 저장, 관리하기 위한 기법은 크게 스키마-의존, 스키마-독립적인 방법으로 구분할 수 있다. 스키마-의존적인 기법에서는 XML 데이터의 구조에 따라 테이블들의 모습과 수가 매번 다르게 된다. [1]에서는 이를 위한 여러 inlining 기법을 소개하였다. 스키마-독립적인 방법은 간선 기반, 노드-기반 등의 방법들이 존재한다. 이 중 노드-기반이 가장 효과적인 것으로 알려져 있으며, [2-4,8] 등의 연구에서 이를 기반으로 한 여러 저장 기법들을 보인 바 있다. Lu 등[16]은 XML의 여러 RDBMS로의 저장 기법들을 비교하였다. 노드-기반 저장 기법은 레이블링 기법, 경로 구체화 기법, 기존 인덱스 구조를 그대로 이용할 수 있는 장점을 가진다. 이 노드 기반 저장 기법의 단점은 XML 데이터가 엘리먼트 단위로 나뉘어져 저장되므로, 질의 결과를 반환 시에는 엘리먼트들을 모아 다시 XML로 재구성 시 성능 저하가 있다는 점이다. 이 때문에 가장 최근의 상용 DBMS들, IBM System RX[6], Oracle[7], MS SQL Server 2005[5]에서는 자체 형식(native format)으로 XML을 그대로 저장함으로써 재구성을 피하도록 한다. 하지만 이 경우에도 인덱스를 구축하기 위해서, 노드 기반 저장 기법은 각 엘리먼트와 관계형 모델의 열 개념간의 사상을 위해 그대로 이용된다. 특히 [5]에서는 자체 형식으로 저장된 XML이 개념적으로 하나의 테이블로 보이고, 여기에 관계형 인덱스를 구축하여 질의 처리하는 방법을 보인다. 가장 최근에는 Grust 등[10]이 기존의 관계형 DBMS의 B+-tree의 효율적인 이용을 통해 자체 포맷으로 XML을 저장하는 데이터베이스보다 더 좋은 질의 처리 성능을 가질 수 있음을 보였다. 비트맵 인덱스는 상대적으로 높은 인덱스 갱신 비용을 감수하면서, 효과적인 논리 연산자의 지원과 빠른 질의 처리를 보장 받기 위한 방법으로 데이터베이스, 특히 OLAP과 의사결정 지원 시스템(decision supporting system) 업무에 널리 이용되고 있다. 비트맵 인덱스는 특히 적은 수의 유일 값(적은 수의 애트리뷰트 카디널리티)을 가지는 컬럼을 색인하고, 논리 연산자를 지원하는데 있어 좋은 방법으로 간주되었다[15]. 최근에는 여러 압축 기법과 인코딩 기법들을 통해 보다 높은 카디널리티를 갖는 관계형 데이터를 효과적으로 지원한다. 높은 애트리뷰트 카디널리티에 대해서 비트벡터는 런-길이 인코딩 기법으로 압축될 수 있다. WAH[21]는 비트열-논리 연산자가 시스템에서 워드 단위로 수행된다는 점에 착안하여, 비트열-논리 연산자를 압축 상태에서도 효과적으로 지원하기 위해 리터럴 워드와 압축 워드들을 혼재하도록 고안된 압축 기법이다. Chan 등[25]은 비트맵 인덱스의 비트벡터를 하나의 유일한 값에 대응 시키는 동등 인코딩(equality) 방법 이외에 영역 인코딩(range encoding)

과 구간 인코딩(interval encoding)과 같이 값-도메인에 대응하는 비트벡터의 수를 줄이는 방법과 이를 이용한 여러 질의 처리 방법을 고안하였다[25]. *bitCube*[26]는 비트맵 인덱스를 XML 질의 처리에 이용한 첫 번째 연구이다. 여기에서 저자들은 2차원의 비트맵 인덱스를 이용하여 X축의 비트벡터는 선형 경로를, Y축의 비트벡터는 텍스트 값을 대표하도록 하여 경로-값 질의에 비트맵 인덱스를 이용할 수 있는 방법을 제안하였다. 하지만 저자들은 가지 패턴 질의에 대한 지원 방법에 대한 고려는 하지 않았으며, 이 인덱스는 압축되기 어려워 인덱스 크기가 큰 단점이 존재한다. Kaushik 등[22]은 XML 가지 패턴 질의에 대한 커버링 인덱스 개념을 처음으로 소개하였고, 가지 패턴 질의에 대한 인덱스의 표현력과 인덱스 크기의 장단점에 대하여 논하였다. 하지만, 여기에서의 인덱스는 인덱스 그래프의 일종인 F&B 인덱스로, 경로-구체화와 노드 기반 저장 기법을 기반으로 하는 본 연구와는 차이가 있다.

8. 결론 및 향후 연구

이 논문에서는 여러 비트맵 인덱스들을 이용하여 관계형 테이블에 저장된XML 문서에 대한 가지 패턴 질의 처리를 지원하는 여러 기법들을 소개하고, 이들에 대한 성능을 분석하였다. 여기에서는 커서 기능이 삽입된 비트맵 인덱스를 이용하여 관계형 DBMS에 저장된 XML 데이터에 대하여 가지 패턴 질의 처리를 지원한다. 커서와 질의 처리 알고리즘은 압축된 비트벡터 상에서 압축해제 없이 동작하는 이점을 가진다. 논문에서 제안하는 6가지 인덱스는 실험을 통하여 기존 휼리스틱 가지 조인 기법과 비교하여 장단점을 논하였다. bitTwig 기법은 트리 구조가 깊지 않은 XML 데이터에 대하여 다른 기법들에 비해 우월한 성능을 보였다. 문서구조가 깊고 재귀적인 XML 데이터에 대해서는 advance(q, k) 를 이용한 데이터 읽기를 건너뛰기 할 수 있는 태그 이름 기반의 비트맵 인덱스가 가장 좋은 성능을 보였다. 향후 연구로는 재귀적인 XML 데이터에 대하여 효과적인 경로 문자열 검색을 수행할 수 있는 기법을 고안함으로써 깊은 트리의 XML 문서에 대해 비트맵 인덱스에서 유효한 비트벡터의 발견을 보다 빠르게 하는 방법이 있다. 또한 XML 문서의 갱신에 따른 비트맵 인덱스의 인덱스 재구축 비용의 최소화가 향후 연구로 남아 있다.

참 고 문 헌

[1] J. Shanmugasundaram, H. Gang, K. Tufte,, C. Zhang, D.J. DeWitt & J. Naughton, "Relational databases for querying XML documents: Limitations and opportunities," *In Proceedings of the International Conference on Very Large Data Bases*, pp.302-314, 1999.

[2] M. Yoshikawa, T. Amagasa, T. Shimura, & S. Uemura, "XRel: a path-based approach to storage and retrieval of XML documents using relational databases," *ACM Transactions on Internet Technology*, 1(1):110-141, 2001.

[3] C. Zhang, J. Naughton, D. DeWitt, Q. Luo, & G. Lohman, "On supporting containment queries in relational database management systems," *In Proceedings of the 2001 ACM SIGMOD international conference on Management of data*, pp.425-436. ACM Press New York, NY, USA, 2001.

[4] P.J. Harding, Q. Li, & B. Moon, "XISS/R: XML indexing and storage system using RDBMS," *In Proceedings of the 29th international conference on Very large data bases*, pp.1073-1076. VLDB Endowment, 2003.

[5] S. Pal, I. Cseri, O. Seeliger, G. Schaller, L. Giakoumakis, & V. Zolotov, "Indexing XML data stored in a relational database," *In Proceedings of the 30th international conference on Very large data bases*, pp.1146-1157, 2004.

[6] K. Beyer, and Others, "System RX: one part relational, one part XML," *In Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, pp.347-358, ACM New York, NY, USA, 2005.

[7] Z.H. Liu, M. Krishnaprasad & V. Arora, "Native XQuery processing in oracle XMLDB," *In Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, pp.828-833, ACM New York, NY, USA, 2005.

[8] M. Rys, D. Chamberlin, & D. Florescu, "XML and relational database management systems: the inside story," *In Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, pp.945-947, ACM New York, NY, USA, 2005.

[9] H. Lu, J.X. Yu, G. Wang, S. Zheng, H. Jiang, G. Yu, & A. Zhou, "What makes the differences: benchmarking XML database implementations," *ACM Transactions on Internet Technology (TOIT)*, 5(1): 154-194, 2005.

[10] T. Grust, J. Rittinger, & J. Teubner, "Why off-the-shelf RDBMSs are better at XPath than you might expect," *In Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, pages 949-958. ACM Press New York, NY, USA, 2007.

[11] G. Gou & R. Chirkova, "Efficiently querying large xml data repositories: a survey," *IEEE Transactions on Knowledge and Data Engineering*, 19(10):1381-1403, 2007.

[12] N. Bruno, N. Koudas, & D. Srivastava, "Holistic twig joins: optimal XML pattern matching," *In Proceedings of the 2002 ACM SIGMOD inter-*

national conference on Management of data, pp.310-321, 2002.

[13] H. Jiang, W. Wang, H. Lu, & J.X. Yu, "Holistic twig joins on indexed XML documents," *In Proceedings of the 29th international conference on Very large data bases*, Volume 29, pp.273-284, 2003.

[14] T. Chen, J. Lu, & T.W. Ling, "On boosting holism in XML twig pattern matching using structural indexing techniques," *In Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, pp.455-466. ACM New York, NY, USA, 2005.

[15] P. O'Neil & D. Quass, "Improved query performance with variant indexes," *In Proceedings of the 1997 ACM SIGMOD international conference on Management of data*, pp.38-49, 1997.

[16] J. Lu, T. Chen, & T.W. Ling, "Efficient processing of XML twig patterns with parent child edges: a look-ahead approach," *In Proceedings of the 13th ACM international conference on Information and knowledge management*, pp.533-542, 2004.

[17] H. Jiang, H. Lu & W. Wang, "Efficient processing of XML twig queries with OR-predicates," *In Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, pp.59-70, 2004.

[18] T. Yu, T.W. Ling & J. Lu, "twigStackList ¬: A holistic twig join algorithm for twig query with not-predicates on XML data," *LECTURE NOTES IN COMPUTER SCIENCE*, 3882:249-264, Springer, 2006.

[19] R. Kaushik, R. Krishnamurthy, J.F. Naughton & R. Ramakrishnan, "On the integration of structure indexes and inverted lists," *In Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, pp.779-790, 2004.

[20] B. Choi, M. Mahoui & D. Wood, "On the optimality of holistic algorithms for twig queries," *LECTURE NOTES IN COMPUTER SCIENCE*, pp.28-37, Springer, 2003.

[21] K. Wu, E. Otoo, & A. Shoshani, "On the performance of bitmap indices for high cardinality attributes," *In Proceedings of the 30th international conference on Very large data bases*, pp.24-35. VLDB Endowment, 2004.

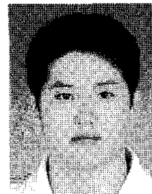
[22] R. Kaushik, P. Bohannon, J. Naughton & H. Korth, "Covering indexes for branching path queries," *In Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, pp.133-144, 2002.

[23] Scientific Data Management Group, University of California Berkeley Lab, "FastBit: An Efficient Bitmap Index Technology," *https://sdm.lbl.gov/ fastbit/*

[24] J. Lu, T.W. Ling, C.Y. Chan, & T. Chen, "From region encoding to extended dewey: on efficient processing of XML twig pattern matching," *In Proceedings of the 31st international conference on Very large data bases*, pp.193-204, 2005.

[25] C. Chan & Y. Ioannidis, "An efficient bitmap encoding scheme for selection queries," *In Proceedings of the 1999 ACM SIGMOD international conference on Management of data*, pp.215-226, 1999.

[26] J.P. Yoon, V. Raghavan, V. Chakilam, & L. Kerschberg, "BitCube: a three-dimensional Bitmap indexing for XML documents," *Journal of Intelligent Information Systems*, 17(2):241-254, 2001.



이 경 하

1998년 충남대 정보통신공학과 학사. 2000년 충남대 정보통신공학과 석사. 2006년 충남대 컴퓨터공학과 박사. 2006년~2007년 한국전자통신연구원 연구원. 2007년~2008년 7월 충남대학교 SW연구소 전임연구원. 2008년 8월~현재 University of Arizona Postdoc. 관심분야는 DBMS, XML, 스트림 데이터, 클라우드 컴퓨팅 등

문 봉 기

1996년 미국 메릴랜드대 박사학위 취득. 현 University of Arizona, Dep't of Computer Science 교수. 현 IEEE TKDE 부편집인. ACM TOIT, JDM 편집진 역임. ACM SIGMOD (2010, 2003, 2001, 1999), VLDB (2010, 2008, 2006, 2001), ICDE (2011, 2009, 2007, 2005), ACM CIKM (2010, 2006), EDBT (2002), ISDB (2002), WWW (2002) 프로그램 위원회 위원 역임. 2002년 ACM SIGMOD Proceedings Chair 역임. 관심분야는 Flash memory, XML indexing/query processing, Temporal/Spatial and Multi-dimensional Databases, Data warehousing, Parallel & distributed processing 등

이 규 철

1984년 서울대학교 컴퓨터공학 학사. 1986년 서울대학교 컴퓨터공학 석사 1990년 서울대학교 컴퓨터공학 박사 1989년~현재 충남대학교 컴퓨터공학과 교수. 1994년 미국 IBM Almaden Research Center 초빙연구원. 1995년 8월~1996년 8월 미국 Syracuse University, CASE Center 초빙 교수. 2000년 2월~2004년 2월 산업자원부 한국 ebXML 전문위원회 위원장. 2001년~현재 전자상거래 표준화 통합 포럼 전자거래 기반 기술위원회 위원장. 2003년 3월~현재 한국전자거래학회 편집 이사. 2005년 1월~현재 한국기록관리학회 이사. 관심분야는 데이타베이스, XML, 웹 서비스, 정보 통합, e-비즈니스, 유비쿼터스 컴퓨팅 등