단축-경로와 확장성 해싱 기법을 이용한 경로-지향 질의의 평가속도 개선 방법

박 희 숙† · 조 우 현††

요 약

최근 인터넷의 폭발적인 성장과 인기로 인하여 인터넷을 통한 정보 교환이 급격으로 증가하고 있다. 또한 XML은 인터넷상에서 데이터를 교환하기 위한 표준인 동시에 중요한 수단이 되고 있다. 따라서 XML 문서를 검색하는데 있어서, 경로-지향 질의에 대한 평가 속도를 개선하는 문제는 중요한 이슈이다. 본 논문에서는 문서 데이터베이스에서 경로-지향 질의의 검색 성능을 개선하기 위한 새로운 인덱싱 방법을 제안한다. 새로운 인덱싱 방법에서는 경로-지향 질의를 효과적으로 수행하기 위해 단축-경로 파일을 생성하고 그것의 해시 코드 값을 인덱스 키로 사용한다. 또한 검색 평가 속도를 촉진시키기 위하여 단축-경로 파일을 확장성 해싱 기법과 결합하는 것으로 검색질의의 평가속도를 가속화하였다.

A Way to Speed up Evaluation of Path-oriented Queries using An Abbreviation-paths and An Extendible Hashing Technique

Hee-Sook Park† · Woo-Hyun Cho††

ABSTRACT

Recently, due to the popularity and explosive growth of the Internet, information exchange is increasing dramatically over the Internet. Also the XML is becoming a standard as well as a major tool of data exchange on the Internet, so that in retrieving the XML document, the problem for speeding up evaluation of path-oriented queries is a main issue. In this paper, we propose a new indexing technique to advance the searching performance of path-oriented queries in document databases. In the new indexing technique, an abbreviation-path file to perform path-oriented queries efficiently is generated which is able to use its hash-code value to index keys. Also this technique can be further enhanced by combining the Extendible Hashing technique with the abbreviation-path file to expedite a speed up evaluation of retrieval.

키워드 : 확장성 해싱(Exdendible Hashing), 단축-경로(Abbreviation-path), 트라이(Trie), XML 문서(XML Document), 경로-지향적 언어(Path-oriented Query Language)

1. 서 론

최근 인터넷의 폭발적인 성장과 인기는 인터넷상에서 정보를 교환하는데 있어 그 중요성이 더욱 증가되고 있다. W3C의해 제안 된 XML(XML : eXtensible Markup Language)은 인터넷상에서 데이터를 교환하기 위한 표준 수단으로서 점차적으로 그 위치를 더욱 확고히 하고 있다. XML 문서에 대한 효과적인 질의 평가 속도 개선에 관한 문제 또한 중요한 연구과제가 되고 있다. 경로-지향적 질의 언어(Path-oriented Query Language)의 한 종류인 XPath(XML Path Language)는 XML 문서의 부분들을 다루기 위한 언어이다. 또한 XPath는 XSLT와 XPointer가 모두 사용하도록 설계되었다[4].

본 논문에서는, XPath와 같은 경로-지향 질의가 입력될 때 구조적으로 저장된 XML문서에 대한 질의 평가 속도를 효과적으로 개선하기 위한 새로운 인덱싱 기술을 제안한다.

새로운 인덱싱 기술은 단축-경로 파일(abbreviation-path file)을 생성하여 이들을 인덱스 키로 사용한다. 그것은 XML 문서의 경로(path)상에 존재하는 각 엘리먼트(Element)들의 첫 번째 문자들을 연결하여 단축-경로를 생성하고 이들을 단축-경로 파일에 저장한다. 저장된 단축-경로들은 인덱스 키로 사용되며 질의 평가속도를 개선하기 위해 단축-경로들과 확장성 해싱(EH : Extendible Hashing)기법을 결합한다. 제안된 인덱싱 기술은 기존의 제안된 인덱싱 기술들에서 제안한 방법보다 질의 평가속도와 요구되는 기억 공간측면에서 더욱 향상된 결과를 나타낸다. 그 이유는 확장성 해싱은 키의 삽입이나 삭제 연산에서 기억공간의 요구량이 동적으로 변하기 때문이다.

본 논문은 다음과 같이 구성된다. 2장에서 관련 연구를 기술

† 준 회 원 : 부경대학교 대학원 전자컴퓨터정보통신공학부
†† 종신회원 : 부경대학교 전자컴퓨터정보통신공학부 교수
논문접수 : 2004년 2월 23일, 심사완료 : 2004년 8월 26일

하며, 3장에서는 기존의 알고리즘 소개 및 문제점을 제시하고, 4장에서 제안된 시스템의 설계 및 구현에 관하여 기술하며, 5장에서는 기존의 방법과 제안된 방법의 성능을 비교 분석하고, 마지막으로 6장에서 결론 및 향후 연구과제를 기술한다.

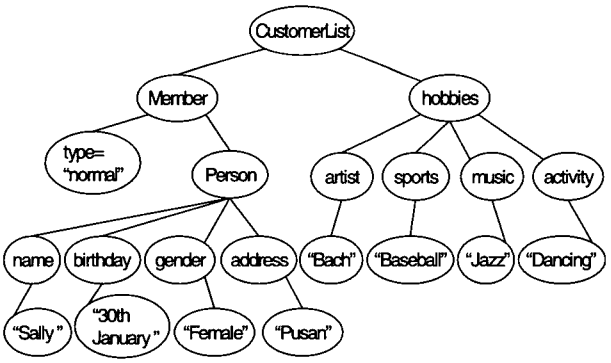
2. 관련 연구

2.1 XML 관련기술

XML은 SGML에서 유도되었으며, 단순하고 매우 유연한 텍스트 형식을 가진다. XML문서는 시작태그와 끝 태그로 둘러싸여 있으며, 태그는 대소문자를 구별한다. 또한 XML문서는 DTD(Document Type Descriptor)에 의해 미리 기술된 구조와 태그 규칙을 따라야만 한다. 문법에 맞게 작성된 XML문서를 "적정 문서(well-formed document)"라 하며, XML문서가 문법에 맞으면서 DTD에 맞게 작성된 문서를 "유효한 문서(valid document)"라 하고 이것을 체크하기 위한 프로그램을 파서(parser)라 한다[7, 10]. XML문서는 DOM(Document Object Model)파서를 이용하여 트리(tree)와 같이 표현될 수 있다. 트리에서 노드의 형태는 각각 엘리먼트(Element)와 애트리뷰트(Attribute) 그리고 텍스트(Text)중에 한 가지이다[1, 5]. (그림 1)은 XML문서와 XML문서의 트리 구조를 보여주고 있다[5, 10].

```
<?xml version = "1.0"?>
<!DOCTYPE CumtomeList>
<CustomerList>
   <Member type = "normal">
     <Person>
        <name>Sally</name>
        <birthday> 30th January </birthday>
        <gender> Female </gender>
        <address> Pusan </address>
     </Person>
   </Member>
   <hobbies>
        <artist>Bach</artist>
        <sports> Baseball </sports>
        <music> Jazz</music>
        <activity>Dancing </activity>
   </hobbies>
</CustomerList>
```

(a) XML문서



(b) XML문서의 트리 구조

(그림 1) XML문서와 트리구조

XML문서는 다음과 같은 종류의 노드들로 구성되어 있다.

① 엘리먼트 노드 : 엘리먼트 노드는 레이블로서 엘리먼트의 이름을 가지며 XML문서의 가장 기초를 구성하는 틀이다. 시작태그와 끝 태그는 반드시 일치하여야 하며, 0개이상 엘리먼트, 애트리뷰트, 텍스트 중 한 가지 형태의 자식노드(child node)를 가질 수 있다[5, 10].

② 애트리뷰트 노드 : 애트리뷰트 노드는 레이블로서 애트리뷰트의 이름과 값을 가진다. 엘리먼트의 특성을 표현하기 위해 쓰여지는 정보를 의미한다. 애트리뷰트 노드는 자식노드를 가지지 않는다. 만일 다수의 애트리뷰트가 나타난다면 그들 순서에는 상관이 없다[5, 10].

③ 텍스트 노드 : 텍스트 레이블로서 스트링을 가진다. 텍스트 노드는 자식노드를 가지지 않으며 보통 파싱이 되지 않는다.

2.2 경로-지향적 언어

트리 구조로 표현 가능한 XML문서를 질의하기 위하여 XML-QL, XPath, XQuery 같은 몇 가지 경로-지향 언어들이 제안되었다. XML-QL은 조인과 같은 데이터 조작을 위해 특정 연산들을 가지며 XML데이터 변환을 지원한다. 이것은 새로운 문서 생성과 문서의 부분을 추출하기 위해 트리-브라우징과 트리-변환 연산자를 제공한다. XQuery는 명령라인으로 표현되어지며 경로 연산자(path operator)를 사용하여 엘리먼트 타입들을 연결한다. 경로 연산자에는 '/'와 '//'가 있다. 여기서 '/'는 자식노드를 구별하기 위한 연산자이며 '//'는 후손노드를 구별하는 연산자이다. 또한 심볼'@'를 사용할 수 있으며 이것은 애트리뷰트 이름 앞에 사용한다. XQuery의 단순경로는 다음과 같이 Backus-Naur Form 문법으로 기술한다[1, 6].

```
<simple path> ::= <PathOP><SimplePathUnit>|
        <PathOP><SimplePathUnit> '@'<AttName>
<PathOP> ::= '/' | '//'
<SimplePathUnit> ::= <ElementType>|<Element Type>
                     <PathOP><SimplePathUnit>
```

다음은 단순경로 질의의 한 가지 예이다.

```
<SimplePathUnit> : /CustomerList/Member/Person[gender$contains
                   $'Female']
```

여기서 /CustomerList/Member/Person은 경로이며, [gender$containes$'Female']는 술어(predicate)이다. 즉 엘리먼트 "gender"가 단어 "Female"을 포함하는지 아닌지를 질의한다.

XPath의 기본 목적은 XML문서의 부분들을 다루기 위한 것이다. 또한 문자열과 숫자, 불리언들을 조작하기 위한 기초 기능들을 제공한다. XPath는 XML문서내의 네비게이션을 명세하기 위한 언어이다. 주어진 XML문서에 대한 XPath 표

현의 평가 결과는 문서의 순서에 따라서 정렬된 노드들의 집합이다. XPath 표현은 다음과 같은 문법 구조를 사용한다.

```
LocationPath ::= RelativeLocationPath | AbsoluteLocationPath
AbsoluteLocationPath ::= '/' RelativeLocationPath? | AbbreviatedAbsoluteLocationPath
RelativeLocationPath ::= Step| RelativeLocationPath '/' Step | AbbreviatedRelativeLocationPath
Path ::= /Step1/Step2/···/Stepn
Step ::= AxisSpecifier NodeTest Predicate* | AbbreviatedStep
```

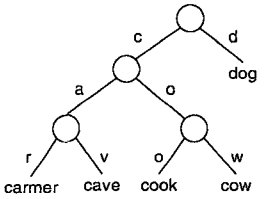
XPath 표현은 왼쪽 Step부터 순차적으로 평가한다. XPath Step은 단일노드에 적용된 결과 노드들의 집합을 선택한다. 각각의 결과 노드 집합은 다음 Step 평가를 위한 컨텍스트 노드(context node)로 사용된다. XPath 표현의 평가 결과는 마지막 스텝에서 선택된 노드집합들의 합집합이다. 스텝내의 AxisSpecifier는 문서에서 네비게이트 되어야 하는 방향을 나타내며 XPath는 child, descendant, parent, ancestor, following-sibling등 12가지를 제공한다. 축(Axes)에 의해서 선택된 노드들은 노드 테스트에 의해 걸러진다. 가장 공통적으로 사용되는 노드 테스트는 노드이름이다. XPath는 대괄호([ ])로 둘러싸인 술어를 포함할 수 있으며 술어는 위치경로의 일부로서 검색한 노드들을 걸러내는 부울식을 말한다. XPath 표현을 사용한 질의 예는 다음과 같다[1, 4].

```
/CustormerList/hobbies/activity[.='Dancing']/@class
```

여기서 /CustormerList/hobbies/activity는 Step들이며 [.='Dancing']는 술어를 나타낸다. 또한 @class는 애트리뷰트이다.

2.3 트라이

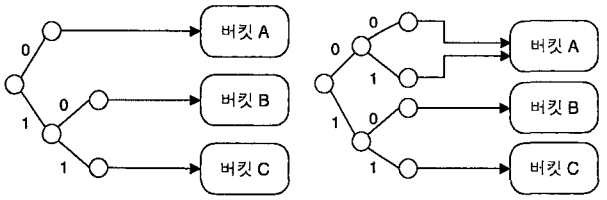
트라이(Trie)는 기수 검색(radix searching)이라고도 불리는데, 이는 탐색 트리의 분기 계수(branching factor)가 키의 각 위치에 나타날 수 있는 서로 다른 기호들의 개수와 같기 때문이다. 트라이는 외부노드인 정보노드(information node)와 중간노드인 분기노드(branch node)로 구성된다. 모든 정보는 외부노드에만 저장되며 분기노드는 정보는 없고 링크만을 가진다. 다른 검색 트리들이 입력되는 노드의 순서에 따라 모양이 변하는 것과 달리 트라이는 입력순서에 관계없이 항상 같은 모양을 나타낸다. 트라이의 정보노드는 서로 구별할 수 있는 자리의 수만큼의 깊이(depth)에 위치한다[2, 13]. 예를 들어 cow, cook, cave, carmer, cost, dog등의 키를 저장하는 트라이를 만들면 (그림 2)와 같다.



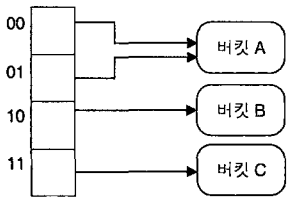
(그림 2) 트라이의 예

2.4 확장성 해싱

확장성 해싱에서의 아이디어는 기존의 해싱을 트라이라는 검색방법과 결합하는 것이다. 확장성 해싱의 기본적인 접근 방법은 기수가 2인 트라이를 이용한다. 검색 결정은 비트 대 비트 단위로 일어난다[2, 13]. 트라이가 외부노드에 한 개의 정보만을 저장하는데 비해 확장성 해싱의 외부노드는 다수의 정보를 저장할 수 있는 버킷을 가진다. 버킷은 다수의 정보를 저장하기 위해 각 버킷마다 고정된 수의 슬롯(slot)을 가진다. 슬롯의 수에 따라 한 개의 버킷에 저장될 수 있는 최대 정보의 수가 결정된다. 확장성 해싱에서 각 외부노드는 실제 데이터의 검색키가 저장된 버킷의 주소를 저장한다. 확장성 해싱의 구성요소는 디렉토리와 버킷(directory, bucket)의 쌍으로 구성되며 디렉토리의 확장이나 축소 그리고 페이지 분할, 병합을 통하여 동적인 검색 구조를 형성한다. 확장성 해싱을 구성하기 위한 과정은 다음과 같다. 먼저 각 키에 대한 해시코드를 구한다. 그런 다음 구해진 해시코드를 이용하여 이진 트라이를 구성하고 이진 트라이를 완전이진 트리(complete binary tree)로 변형한다. 마지막으로 완전이진 트리를 디렉토리로 변환한다. (그림 3)은 디렉토리의 깊이가 2인 트라이와 완전이진 트리를 보여주고 있다. 따라서 디렉토리내의 셀의 수는 2^2, 즉, 4개가 되며 정보를 저장하기 위한 버킷을 결정하는데 2비트의 해시코드 값을 사용한다. (그림 3)(b)는 (그림 3)(a)의 이진 트라이를 완전이진 트리로 구성한 것이며 (그림 3)(c)는 완전이진 트리를 디렉토리로 변환한 것이다. (그림 3)(c)에서 해시주소가 0으로 시작하는 키 즉, 00과 01로 시작하는 해시주소를 가지는 키들은 버킷 A에 저장되고 해시주소가 10으로 시작하는 키는 버킷 B에 그리고 11로 시작하는 주소를 가진 키는 버킷 C에 저장된다. 버킷 A의 버킷 깊이는 0, 버킷 B는 10, 그리고 버킷 C는 11이다.



(a) 2진 트라이 (b) 완전이진 트리

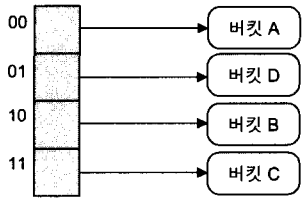


(c) 디렉토리

(그림 3) 확장성 해싱의 예

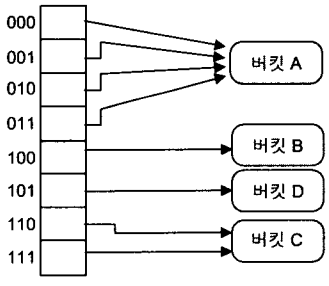
2.5 버킷의 오버플로 처리

어떤 해싱 시스템에서 핵심 이슈는 버킷에 오버플로가 일어났을 때 어떻게 처리하는가에 대한 문제이다. 확장성 해싱 시스템은 오버플로가 발생한다면 오버플로와 관련한 주소 공간을 증가시키는 방법을 사용한다. 확장성 해싱에서 실제 각 버킷들은 선형적으로 검색되어야 한다. 만약 (그림 3)(c)의 버킷 A에 새로운 레코드를 삽입하려고 할 때 오버플로가 일어났다고 가정한다면 디렉토리 주소 00과 01이 동일한 버킷 A를 가리키고 있으므로 분할이 가능하다. 따라서 (그림 4)와 같이 새로운 버킷 D를 만들고 버킷A에 혼합되어 저장되어 있는 00과 01로 시작하는 버킷주소를 가진 키들 중에서 01로 시작하는 주소를 가진 키들은 모두 버킷 D로 이동을 한다[2].



(그림 4) 버킷 A분할후 디렉토리구조

이번에는 (그림 3)(c)에서 버킷 B에 새로운 레코드를 삽입하려고 할 때 오버플로가 일어났다면 더 이상 버킷 분할이 불가능하기 때문에 디렉토리의 확장이 요구된다. 따라서 디렉토리 주소에 사용되는 비트수를 2비트에서 3비트로 증가한다. 결과적으로 디렉토리 셀의 수는 2^3개이므로 전체 디렉토리 셀의 수는 현재의 2배로 늘어난다. 중간 과정에서 생성되는 트라이와 완전이진 트리의 깊이 또한 2에서 3으로 늘어난다. 확장 이후 디렉토리의 구성은 (그림 5)와 같다. (그림 4)에서 버킷 A와 버킷 D 그리고 버킷 B와 버킷 C를 버디 버킷(buddy bucket)이라 하며 이들은 삭제 연산에서 각각 단일 버킷으로 통합이 가능하다.



(그림 5) 디렉토리 확장 후

3. 기존 알고리즘의 소개 및 문제점

기존 Chen의 논문[1]은 경로-지향 질의에 대한 평가 성능을 향상시키기 위하여 경로 서명 파일(path signature file)과 패트리샤 트리(PT : Patricia Tree)를 결합한 검색 알고리즘을 제안하고 있다. 경로 서명 파일내의 각 경로 서명을 생성하기 위해 사용되는 서명 값은 1로 설정된 m개의 비트를 가진 길이 F인 해시코드화 된 비트패턴으로 이루어져 있으며 경로서명 파일을 생성하기 위해 중첩기호법을 사용한다. 경로서명 파일은 부정확한 여과장치(inexact filter)의 개념을 기반으로 하고 있다. 따라서 그들은 많은 부적합한 값들을 버리는 빠른 테스트를 제공하지만 적합한 값들은 확실히 테스트를 통과한다. 그러나 몇몇의 값들은 실제로는 검색 요구 조건을 만족하지 않는다 할지라도 여과장치를 통과하는 경우도 있다. 이들을 허위드롭(False Drop)이라 한다. 허위드롭의 가능성을 최소화하기 위해서는 다음의 공식을 사용한다.

$$F_d = 2^{-m} \quad (1)$$

$$F\ln 2 = mD \quad (2)$$

위의 공식에서 F_d는 허위드롭 가능성을 나타내며, F는 서명의 크기를, m은 각 서명에 1로 설정된 비트 수를, D는 논리블럭의 길이 즉 단어의 수를 나타낸다. 예를 들어 위의 (그림 1)(b)에서 'CustomerList'과 'Member'의 경로서명을 생성하여 보자. 만약 F를 12로 하고 D를 2로 결정한다면 위의 공식 (1)에 의해 m의 값은 4가 된다. 따라서 두 개의 엘리먼트에 대한 경로서명은 엘리먼트 'CustomerList'와 'Member'의 서명을 중첩기호법으로 연산한 것으로 다음과 같다.

CustomerList		: 001 000 110 001
Member	∨	: 000 010 110 100
경로서명		: 001 010 110 101

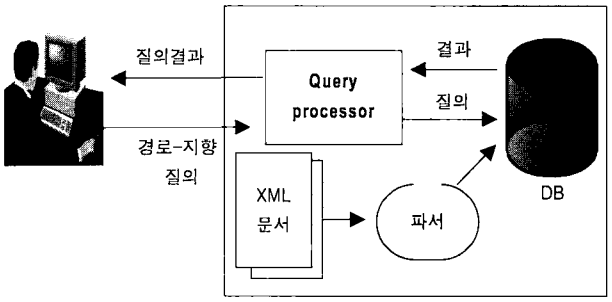
경로서명파일에서 한 개의 경로서명의 길이 F가 주어졌을 때 비트 패턴 값들 중에서 1로 설정 된 비트의 수가 거의 50%를 유지할 때 허위드롭의 가능성이 최소화된다. 만약 m의 값이 50%를 넘는다면 검색 비교 횟수는 줄어들지만 허위드롭의 가능성 증가로 성능이 저하된다. 반대로 m의 값이 50%보다 더 작아지게 된다면 검색 비교횟수가 늘어나게 된다.

패트리샤 트리는 이진 디지털 트리(binary digital tree)이며 내부노드는 분기를 위한 각 비트의 위치를 결정하는 값을 저장하고 각 외부노드에는 한 개의 정보만을 저장한다[14]. 기존의 논문에서 정보검색 연산은 패트리샤 트리에서 테스트 되어지는 경로서명에서 각 위치 비트의 값이 1이면 오른쪽 자식노드를 따라가고 0이면 양쪽 자식노드를 모두 따라가게 된다. 따라서 입력되는 경로-지향 질의의 경로서명 해시코드가 포함하는 1의 비트수에 따라서 검색성능이 많은 영향을 받는다. 만약 1의 비트수가 많아지면 검색을 위한 비교횟수는 적어지고 1의 비트의 수가 적으면 비교횟수가 많아지게 된다. 또한 입력되는 인덱스 키의 수가 많아짐에 따

라 패트리샤 트리 구조 또한 확장이 요구된다. 그 이유는 패트리샤 트리는 외부노드에 단일의 키 값만을 저장하기 때문이다. 따라서 대규모 데이터베이스에서 사용하기에는 적합하지 않다. 기존의 논문에서 제안한 검색방법은 키의 검색을 위해 최대 $O(N/2^{l})$의 비교횟수를 요구한다. 여기서 N은 경로서명파일이 포함하는 경로서명의 수이며 l은 입력되는 경로-지향 질의에 포함된 경로서명에서 2진수 1로 설정된 비트들의 개수이다.

4. 시스템 설계 및 구현

본 논문에서 구현한 시스템의 논리적인 전체 구성도는 (그림 6)과 같다. 시스템에서 질의처리기(Query processor)는 사용자가 입력한 경로-지향 질의를 수행하기 위해 관계형 질의어로 변환하고 질의어 처리결과를 사용자에게 전달하기 위한 출력양식을 만들어 주는 역할을 수행한다. XML문서는 파서에 의해 파싱되어 관계형 데이터베이스에 엘리먼트,텍스트, 애트리뷰트 테이블로 각각 분리 저장된다. 엘리먼트 테이블은 {문서ID(숫자), 엘리먼트ID(숫자), 엘리먼트명(문자), 부모ID(숫자), 단축경로(문자)}로 구성되며, 텍스트 테이블은 {문서ID(숫자), 부모ID(숫자), 텍스트값(문자)}로 구성된다. 그리고 애트리뷰트 테이블은 {문서ID(숫자), 부모ID(숫자), 애트리뷰트명(문자), 애트리뷰트값(문자)}로 구성된다.



(그림 6) 시스템의 논리적 구성도

본 논문에서 우리는 새로운 인덱싱 방법을 제안한다. 새로운 인덱싱 방법에서는 인덱스 키 값으로 단축-경로 파일을 사용한다. 단축-경로 파일은 XML문서에서 엘리먼트들의 경로상에 존재하는 각 엘리먼트의 첫 번째 머리글자만을 추출하여 이들을 연결하여 생성한다. 또한 경로-지향 질의의 검색성능을 개선하기 위하여 단축-경로 파일은 확장성 해싱 기법과 결합을 한다. 따라서 본 논문에서 제안한 방법은 확장성 해싱 구조를 구성하기 위해 단축-경로의 해시코드를 인덱스 키로 사용한다. <표 1>은 (그림 1)의 XML문서에 대한 단축-경로 파일의 예를 보여주고 있다. 실제 단축-경로 파일에는 단축-경로만이 저장된다.

<표 1> 단축-경로 파일의 예

위치	Element-Path	단축-경로	해시코드
1	CustomerList	C	0001101000101100
2	CustomerList-Member	CM	0001101001111001
3	CustomerList-Member-Person	CMP	0011100110111001
4	CustomerList-Member-Person-name	CMPn	0011101000100111
5	CustomerList-Member-Person-birthday	CMPb	0011101000011011
6	CustomerList-Member-Person-gender	CMPg	0011101000100000
7	CustomerList-Member-Person-address	CMPa	0011101000011010
8	CustomerList-Member-Person-nationality	CMPn	0011101000100111
9	CustomerList-hobbies	Ch	0001101010010100
10	CustomerList-hobbies-artist	Cha	0100000001111000
11	CustomerList-hobbies-sports	Chs	0100011110000000
12	CustomerList-hobbies-music	Chm	0100010100101000
13	CustomerList-hobbies-activity	Cha	0100000001111000

(그림 7)(a)는 확장성 해싱 구조을 구성하기 위한 중간단계로서 <표 1>의 단축-경로 파일에 대한 완전이진 트리이다. (그림 7)(b)는 완전이진 트리를 디렉토리 구조로 변환한 것이다. 본 논문에서는 디렉토리를 구성하기 위하여 디렉토리 주소를 생성할 때 키의 해시코드를 오른쪽에서부터 추출하여 역순(reversal order)으로 만든다. 그 이유는 해시코드의 비트값 분포가 보다 다양해지므로 인덱스를 구성할 때 오버플로 횟수가 적어지기 때문이다. (그림 7)에서 한 개의 버킷에 최대 저장할 수 있는 키의 수를 3으로 제한한다.

예를 들어 /CumstomerList/hobbies/artist란 경로-지향 질의 입력에 대한 검색 수행은 (그림 7)(b)에서 보여지는 것과 같이 굵은 선을 따라 간다. 먼저 입력된 경로-지향 질의에 포함된 경로의 단축-경로 Cha가 만들어지고 이것에 대한 해시코드(0100010100101000)를 생성한다. 디렉토리의 깊이는 4이다. 따라서 해시코드의 오른쪽 4비트를 추출하여 역순으로 만들면 디렉토리주소 0001이 된다. 해당 디렉토리 주소에 기억된 포인터가 지시하는 버킷을 찾아 버킷에 저장된 데이터를 순차적으로 모두 비교 검색하여 결과로 반환되는 것이 Cha의 주소가 된다. 위의 <표 1>에서와 같이 주소 9와 12처럼 만일 단축-경로가 동일한 경우가 발생한다면 버킷에는 하나만 저장된다. 그러나 실제로는 단축-경로가 동일하다 할지라도 서로 다른 경로를 나타내기 때문에 실제 검색 질의를 수행하는 경우 원치 않는 결과도 함께 나타난다. 이것이 본 논문에서 제안하고 있는 인덱싱 방법의 문제점중의 하나이다. 질의처리기는 다음과 같은 질의를 생성한다.

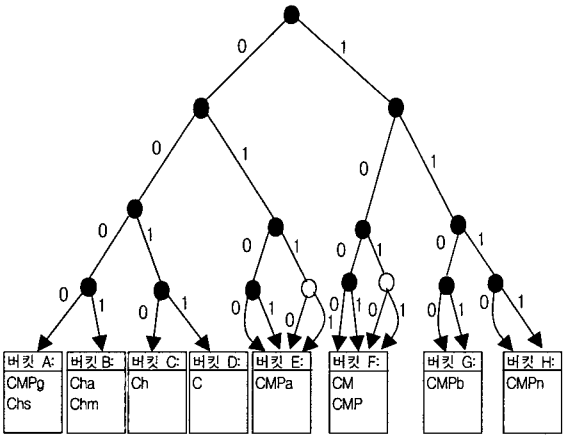
```
SELECT * FROM ElementTable e
WHERE e.단축경로 IN 결과로 반환된 단축-경로 ;
```

위의 질의 수행결과 엘리먼트 테이블에서 단축-경로가

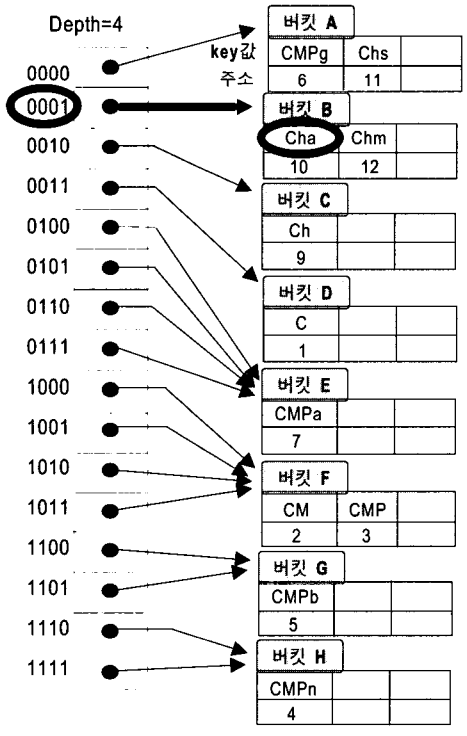
Cha인 레코드는 모두 검색된다. 따라서 실제 검색 조건을 만족하지 않는 레코드라 할지라도 동일한 단축-경로를 가지는 경우 결과로 나타나게 된다.

만약 질의가 /CumstomerList/hobbies[artist$containes$'Bach']의 형태로 입력된다면 질의처리기는 다음과 같은 질의를 생성한다.

```
SELECT * FROM ElementTable e TextTable t
WHERE e.엘리먼트명 = 'artist'
AND e.단축경로 IN 결과로 반환된 단축-경로
AND e.문서ID = t.문서ID
AND e.엘리먼트ID = t.부모ID AND t.텍스트값그'Bach' ;
```



(a) 완전이진 트리



(b) 디렉토리

(그림 7) 완전이진 트리와 디렉토리

```
input : DOM파서에 의해 파싱된 XML문서의 각노드, 트리에서 노드의 레벨
output : 단축-경로파일, 엘리먼트테이블, 에트리뷰트테이블, 텍스트 테이블

XML문서파싱함수(XML문서의 각 노드, 트리에서 노드의 레벨) {
    현재 노드의 노드타입을 구함 ;
  switch(노드타입) {
    case 문서노드 :
      문서객체 생성 ;
      XML문서파싱함수를 재귀적으로 호출하고 파라메터로
      문서의 루트 엘리먼트와 루트 엘리먼트 노드의 레벨을 전달;
      break ;
    case 엘리먼트노드 :
      부모노드ID 값으로 현재 엘리먼트 노드의 ID값을 저장 ;
      현재 엘리먼트 노드의 ID값을 1증가 ;
      엘리먼트 경로추적을 위해 엘리먼트ID를 배열 IDTrace에 저장 ;
      단축-경로 생성에 사용될 경로상의 엘리먼트명을
      배열 PathTrace에 저장 ;
      루트레벨에서 현재 엘리먼트노드 레벨까지
      loop를 돌면서 배열 PathTrace를 사용하여 단축-경로를 생성 ;
      문서ID, 엘리먼트ID, 엘리먼트명, 부모노드ID, 단축-경로를
      엘리먼트 테이블에 삽입 ;
      생성된 단축-경로를 단축-경로 파일에 저장 ;
      현재 엘리먼트 노드가 포함하고 있는 애트리뷰트 개수만큼
      loop를 돌면서 XML문서파싱함수를 재귀적으로 호출하고
      파라메터로 애트리뷰트 노드와 엘리먼트노드의 레벨을 전달 ;
      현재 엘리먼트노드의 자식노드가 존재한다면 자식노드의
      개수만큼 loop를 돌면서 XML문서파싱함수를 재귀적으로
      호출하고 파라메터값으로 자식노드와 자식노드의 레벨을
      전달 ;
      break ;
    case 에트리뷰트노드 :
      부모노드ID 값으로 현재 엘리먼트 노드의 ID값을 저장하고
      문서ID, 부모노드ID, 애트리뷰트명, 애트리뷰트값을
      애트리뷰트 테이블에 삽입 ;
      break ;
    case 파싱 되지 않은 문자 :
    case 텍스트노드 :
      만약 텍스트노드 또는 파싱되지 않는 문자의 길이가 0이 아니면
      부모노드ID 값으로 현재 엘리먼트 노드의 ID값을 저장하고
      문서ID,부모노드ID,텍스트값을 텍스트 테이블에 삽입한다
      그렇지 않다면
      텍스트 테이블에 삽입하지 않는다
    }
}
```

(그림 8) XML문서 파싱 및 단축-경로 파일 생성 알고리즘

(그림 8)은 XML문서 파싱 및 단축-경로 파일을 생성하는 알고리즘이다. 본 논문의 알고리즘은 DOM 파서를 사용하여 파싱을 한다. 입력되는 XML문서는 파싱을 통해 각 노드의 타입에 따라서 엘리먼트 테이블, 애트리뷰트 테이블 그리고 텍스트 테이블에 각각 저장한다. 노드의 타입이 엘리먼트인 경우에 그들의 단축-경로를 생성하고 이를 단축-경로 파일에 저장한다. 또한 생성된 단축-경로는 엘리먼트 테이블의 단축-경로 필드의 값으로 저장한다.

본 논문에서 인덱싱을 위한 알고리즘은 기존의 구현된 확장성 해싱 알고리즘을 변형하여 사용하였다[2]. 확장성 해싱 알고리즘은 디렉토리 파일과 버킷 파일로 구성한다. 디렉토리 파일에는 버킷의 주소가 저장되며 버킷 파일에는 (키, 주소)쌍으로 정보가 저장된다. 확장성 해싱 알고리즘은 삽입, 삭제, 검색 연산을 수행한다. 삽입연산은 삽입할 키의 해시

주소를 계산하여 키가 저장될 디렉토리 주소를 결정한 다음 버킷의 주소를 결정하고 해당 버킷에 키를 삽입한다. 만약 버킷에 오버플로가 일어난다면 버킷의 분할 연산을 수행하고 키를 재분배한다. 버킷의 분할이 더 이상 불가능하다면 디렉토리를 확장하는 연산을 수행한다.

삭제연산에서 가장 중요한 것은 해당키를 삭제한 후 버디 버킷들을 통합하여 사용되는 메모리의 량을 축소하는 작업이다. 삭제연산은 디렉토리주소와 버킷주소를 이용하여 삭제할 키와 관련한 버킷을 메모리로 로드한 후 해당 버킷을 순차적으로 검색하여 키를 찾는다. 만약 삭제할 키가 존재한다면 버킷에서 해당키의 삭제 작업을 먼저 수행한 다음 더 이상의 버디 버킷 통합이 불가능해질 때까지 버디 버킷의 통합 수행한다. 또한 디렉토리의 크기 축소가 가능하다면 디렉토리 크기를 절반으로 축소하는 작업을 수행한 후 이전 디렉토리의 내용을 축소된 디렉토리로 내용을 조정하여 기억하는 작업을 반복 수행한다.

검색연산은 키 검색을 위한 디렉토리 주소 계산에 입력키와 디렉토리의 깊이를 파라미터 값으로 사용한다. 계산된 디렉토리 주소를 사용하여 키와 관련한 버킷의 주소를 결정한 후 해당 주소의 버킷을 메모리로 로드한다. 버킷내의 모든 엔트리를 순차적으로 비교 검색하여 동일한 키 값이 존재한다면 키의 주소를 반환한다.

5. 실험결과 및 성능 비교 분석

본 논문의 시스템 성능실험은 다음과 같은 시스템 환경하에서 이루어졌다. 실험은 윈도우즈 2000 운영체제를 탑재한 펜티엄4 시스템 상의 메모리 1GB, CPU속도 1.7GHz, HDD 80GB와 데이터베이스 시스템은 Oracle 9i를 사용하여 수행하였다. 본 논문에서 입력되는 경로-지향 질의는 경로를 모두 절대 위치경로(absolute location path)형태로 입력하였으며, 경로-지향 질의 유형은 단순경로들만 입력되는 형태와 술어를 함께 포함하는 형태 2가지로 실험하였다. 실험에 사용한 데이터들은 <표 2>와 같다.

〈표 2〉 실험에 사용한 데이터들

단축-경로 개수	XML 문서의 최대 깊이	해시코드 길이(bit)	평균 엘리먼트 수/문서	XML 문서 파일의 수	슬롯의 수/버킷	경로-지향 질의 유형
1,000	4	16	53	19	90	/CustomerList /hobbies /sports
10,000	4	16	210	48	90	
100,000	4	16	300	334	90	/CustomerList /hobbies [artist$containes$'Bach']
500,000	4	16	400	1,250	90	

실험 결과 버킷 파일과 디렉토리 파일에 소요되는 파일의 크기는 <표 3>과 같다. <표 3>에서 보여지는 것과 같이 디렉토리 파일의 크기가 그다지 크지 않기 때문에 전체 디렉토리가 메인 메모리 상에 모두 저장될 수 있다. 따라서 키 검색을 위한 디렉토리 액세스는 단지 한번만 요구되기 때문에 시간 복잡도는 O(1)이 된다. 왜냐하면 확장성 해싱에서는 오버플로 처리를 위한 별도의 공간을 요구하지 않기 때문이다. 키 검색을 위한 비교 횟수는 최악의 경우에도 O(k)이다. 여기서 k은 하나의 버킷에 저장할 수 있는 최대 레코드의 수 즉, 슬롯의 수를 말한다.

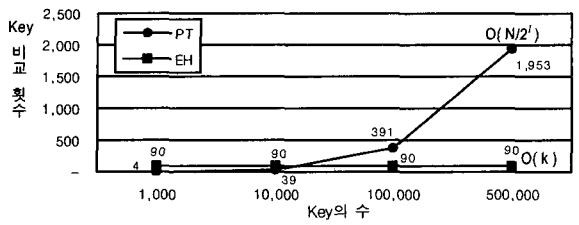
〈표 3〉 디렉토리와 버킷파일의 크기

Key의 수	Directory File의 크기	Bucket File의 크기	슬롯의 수(k) (키의 수/버킷)
1,000	3.2KB	13.6KB	4
10,000	6.02KB	124KB	11
100,000	12KB	1,379KB	24
500,000	48KB	3,656KB	90

실험 결과 확장성 해싱은 입력되는 인덱스 키의 수에는 영향을 받지 않으며 단지 한 개의 버킷이 가지는 슬롯의 수에 따라서 검색 비교 횟수가 달라지게 된다. 따라서 검색은 항상 디렉토리 주소에 의해 결정된 해당 버킷의 내용만을 비교 검색하기 때문에 대규모 데이터베이스에서도 검색이 빠르게 수행된다는 장점을 가진다. 그러나 중간 규모 이하의 데이터베이스 검색에서는 오히려 기존의 방법보다 검색 성능 저하를 나타내며, 단축-경로를 생성할 때 서로 다른 경로에 대해 동일한 단축-경로가 생성되는 문제점을 가지고 있다. 기존[1]의 방법은 중간규모 이하의 데이터베이스에서 제안된 방법보다 우수한 검색 성능을 나타내고 있다. 그 이유는 부적합한 데이터들은 빠르게 검색대상에서 제외되기 때문이다. 그러나 패트리샤 트리 구조는 외부노드에 단 한 개의 정보 노드만이 저장이 가능하기 때문에 인덱스 키의 수가 많아짐에 따라서 경로서명 길의 확장 및 패트리샤 트리 구조를 확장 해야하는 문제점이 있다.

(그림 9)는 기존[1] 인덱싱 방법과 본 논문에서 제안한 새로운 인덱싱 방법에 대한 정보 검색에 요구되는 최대 키 비교 횟수를 분석한 그래프이다. 성능 비교 실험을 위해 기존[1]의 인덱싱 방법 실험에도 동일한 조건의 데이터들을 사용하였으며, 위의 식 (1)과 식 (2)에 의해 경로서명의 크기 F를 16으로 할 때 D는 XML문서의 최대 깊이인 4가 되며 m은 약 3이 된다. 따라서 허위드롭의 가능성을 최소화하고 최대의 성능을 유지하기 위해 l의 값은 8로 결정하였다. (그림 9)에서 보여지는 것과 같이 기존의 인덱싱 기술은 인덱스 키의 수가 증가함에 따라서 검색을 위한 키의 비교 횟수가 비례적으로 증가하고 있다. 그러나 본 논문에서 제안한 새로운 인덱싱 방법은 입력되는 인덱스 키의 수에 관계없이 최악의 경우에도 항상 버킷의 슬롯 수만큼만 비교 연산을 수행하기 때문에 인덱스의 양에 관계없이 항상 일정한 검색 성능을 유지한다.

Extendible Hashing(EH) : $k = 90$
Patricia Tree(PT) : $l = 8$



(그림 9) 패트리샤 트리(PT)와 확장성 해싱(EH)의 검색횟수 비교 분석

6. 결론 및 향후 연구과제

본 논문에서, 우리는 관계형 데이터베이스에 저장된 XML문서에 대한 경로-지향 질의어의 평가속도를 향상하기 위한 새로운 인덱싱 방법을 제안하였다. 제안한 인덱싱 방법은 단축-경로파일과 확장성 해싱 기법을 결합함으로서 입력되는 키의 수에 관계없이 최악의 경우에도 항상 일정한 검색 성능을 나타낸다는 것을 보였다. 우리의 실험에서 인덱스 키의 양에 따라서 하나의 버킷에서 요구되는 최저 슬롯수가 존재함을 알게 되었으며, 최저 슬롯 수 이하로 각 버킷의 슬롯 수를 결정하는 경우에는 인덱스 구성의 효율성이 떨어지는 것으로 나타났다. 따라서 향후 연구과제로는 상대 위치 경로(relative location path) 형태로 입력되는 경로-지향 질의에 대한 처리와 서로 다른 경로가 동일한 단축-경로를 생성하는 경우 효율적인 처리 방안에 대한 연구가 필요하며 인덱스 키의 양에 따른 한 개의 버킷이 가져야할 가장 적정한 슬롯 수 결정에 관한 증명이 요구된다.

참 고 문 헌

[1] Y. Chen and G. Huck, "Path signature : A Way to Speed up Evaluation of Path-oriented Queries in Document Databases," WISE2000, pp.240-244, 2000.

[2] M. J. Folk and B. Zoellick and G. Riccardi, "File Structures An Object-Oriented Approach with C++," Addison-Wesley, 1998.

[3] W. B. Frakes and R. Baeza-Yates, "Information Retrieval : Data Structures and Algorithms," Prentice Hall PTR ; Facsimile edition, 1992.

[4] W3C, "XML Path Language (XPath) 2.0," http://www.w3.rg/2003/08/DIFF-xpath20, 2003.

[5] W3C, "Extensible Markup Language (XML)," http://www.w3.org/XML/, 1998.

[6] W3C, "XML Query (XQuery) Requirements," http://www.w3.org/TR/2003/WD-xquery-requirements-20031112, 2003.

[7] W3C, "Document Object Model (DOM)," http://www.w3.org/DOM/, 2002.

[8] C. Zaniolo et al., "Advanced Database Systems," Morgan Kaufmann Publishers, 1997.

[9] D. H. C. Du and Member and IEEE and S. Tong, "Multilevel Extensible Hashing : A File Structure for Very Large Database," IEEE Trans. on Knowledge and Data Eng., Vol. 3, No.3, 1991.

[10] H. M. Deitel and P. J. Deitel and T. R. Nieto and T. Lin, P. Sadhu, "XML How TO PROGRAM," Prentice Hall, 2000.

[11] S. Helmer and T. Neumann and G. Moerkatte, "A Robust Schema for Multilevel Extendible," Proc. 18th Int. Sym. on ISCIS, Antalya, pp.220-227, 2003.

[12] H. Mochizuki and M. Koyama and M. Shishibori and J. Aoe, "A substring search algorithm in extendible hashing," Informatics and Computer Science : An International Journal Vol.108, Issue 1-4, pp.13-30, 1998.

[13] Rambihia, "XML Distributed Systems Design," SAMS, 2002.

[14] W. Frakes and R. Baeza-Yates, "Information Retrieval : Data Structures and Algorithms," Prentice-Hall, 1992.

[15] C. Faloutsos an ACM Computing Surveys, "Access Methods for Text," Vol.17, No.1, pp.48-74, 1985.

박 희 숙

e-mail : bg007@edunet4u.net

1998년 경남대학교 교육대학원 전자계산 교육전공(교육학석사)

2004년 부경대학교 대학원 컴퓨터공학과 (박사수료)

관심분야 : 객체지향 데이터베이스, 인덱싱 성능개선 문제, 멀티미디어데이터베이스

조 우 현

e-mail : whcho@pknu.ac.kr

1985년 경북대학교 전산공학전공(학사)

1988년 경북대학교 대학원 전산공학전공 (공학석사)

1998년 경북대학교 대학원 전산공학전공 (공학박사)

1988년~1989년 한국전자통신연구소 지능망연구실 연구원

2002년~2003년 텍사스주립대-산마르코스 방문연구

1989년~현재 부경대학교 공과대학 전자컴퓨터정보통신공학부 교수

관심분야 : 지식의 표현과 병렬처리, 멀티미디어 데이터베이스 관리 시스템, 객체 지향 데이터베이스