XML 데이터의 효과적인 검색을 위한 다중 경로 인덱스 기법

(Multi-Path Index Scheme for The Efficient Retrieval of XML Data)

송 하 주　김 형 주

서울대학교 컴퓨터공학과

요약

확장 경로식은 '*' 문자를 이용하여 여러 개의 경로를 간단하게 나타내기 위해 사용하는 것으로 객체지향 데이터베이스(object-oriented database : OODB)에 저장된 XML 데이터를 검색하는 질의를 표현하기에 유리하다. 본 논문은 이러한 확장 경로식을 포함하는 OQL 질의를 효과적으로 처리하기 위한 인덱스 구조로서 다중 경로 인덱스 기법을 제안한다. 제안하는 기법은 확장 경로식에 포함되는 각각의 단일 경로에 대해 고유한 경로 식별자를 부여한다. 그리고 인덱스 키값과 경로 식별자를 조합하여 저장함으로써 하나의 인덱스만을 이용하여 다수의 경로에 대한 검색과 단일한 경로에 대한 인덱스 기능을 동시에 지원하도록 하였다. 이 기법은 확장된 경로식에 대해 기존 인덱스를 여러 개 사용하는 방법보다 검색 성능을 높일 수 있고 B^+-트리 인덱스 구조를 크게 변형하지 않고 사용할 수 있어 실용성이 우수하다.

Abstract

Extended path expressions are used to denote multiple paths concisely by using '*' character. They are convenient for expressing OQL queries to retrieve XML data stored in OODBs. In this paper, we propose a multi-path index scheme as a new index scheme to efficiently process queries with extended path expressions. Our proposed index scheme allocates a unique path identifier for every possible single path in an extended path expression and provides functionalities of both a single path indexing and multiple path indexing through the composition of index key and path identifier while using only a index structure. The proposed index scheme provides better performance than single-path index schemes, and is practical since it can be implemented by little modification of leaf records of a B^+-tree index.

1 서론

인터넷 상의 데이터 교환을 위한 사실상의 표준으로 떠오르고 있는 XML(extensible markup language) [1] 데이터를 효율적으로 저장하고 검색하기 위해 많은 연구가 최근 진행되고 있다. 그에 따라 관계형 데이터베이스(relational database : RDB)나 객체지향 데이터베이스(object-oriented database : OODB)를 이용하여 XML 데이터를 보다 효과적으로 저장하고 검색하기 위한 연구가 활발하다 [2, 3, 4]. 특히 객체지향 데이터베이스는 데이터 모델이 XML의 특성을 자연스럽게 반영할 수 있고, XML에 대한 질의어를 기존 객체 질의어를 확장하여 지원할 수 있기 때문에 XML의 저장 장치로서 각광을 받고 있다. XML에 대한 질의는 기존 객체 질의어에서는 지원되지 않던 기능들을 필요로 하는데 그 중 하나가 객체들간의 참조 관계를 나타내는 경로식의 표현이 더 자유롭다는 점이다. 다음은 그러한 예를 보인 것이다.

그림 1-(a)는 XML 데이터의 스키마에 해당하는 DTD(document type definition)의 한 예를 부분적으로 보인 것이고, 그림 1-(b)는 그림 1-(a)의 XML DTD에 해당하는 XML 문서를 OODB에 저장하기 위한 OODB의 스키마를 나타낸 것이다. 그림 1-(b)에서 네모 칸과 이탤릭체 문자열은 각각 클래스와 클래스의 속성을 나타낸다. XML DTD를 OODB의 스키마로 변환하는 기법에 대해서는 참고문헌[3]에서 제시된 방법을 사용한다. OODB의 스키마를 통해 저장된 XML 데이터에 대해서는 다음과 같이 경로식을 이용한 질의가 가능하다.

- 질의 예1:

  subsection의 본문이 최종적으로 변경된 날짜가 2000년 1월 1일인 서적을 모두 검색하라.

  ```
  select b from b in book
  where b.chapter.section.subsection.body.mdate = '2000:01:01';
  ```

경로식을 포함하는 질의를 처리하기 위해서는 경로식으로 표현된 객체간의 참조 관계를 따라 객체를 차례로 방문해야 하는데 이 과정을 객체 탐색(object navigation)이라고 한다. 객체 탐색은 디스크 상의 객체를 방문하기 때문에 많은 오버헤드를 유발하며, 특히 방문해야 하는 객체들이 디스크 상에 흩어져 있는 경우에는 그 오버헤드가 더욱 증가하게 된다 [5]. 그리하여 객체 탐색을 효과적으로 지원하기 위한 인덱스 구조로 중첩 인덱스(nested index), 경로 인덱스(path index) [6], 다중 인덱스(multi index), ASR(access support relation) [7], 계층 조인 인덱스 [5] 등이 제안되었다.

한편 XML 데이터를 편리하게 검색하기 위해서는 다음과 같이 '*' 를 이용한 경로식을 사용할 수 있다.

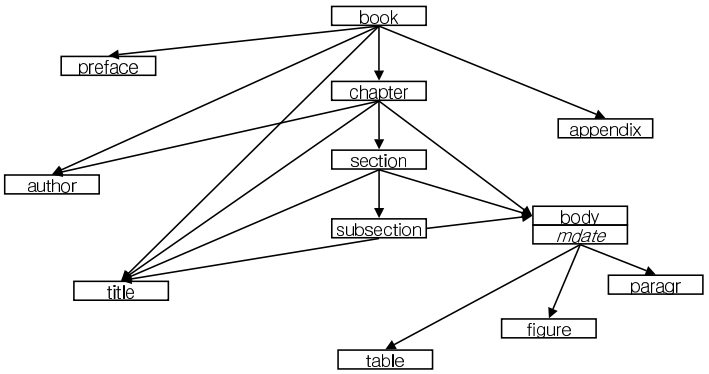
- 질의 예 2:

  본문의 내용이 최종적으로 변경된 날짜가 2000년 1월 1일인 서적을 모두 검색하라.

  ```
  select b from d in book
  ```

```
<!DOCTYPE book [
<!ELEMENT book        (title,author+,preface,chapter+,appendix)>
<!ELEMENT chapter     (title,author*,body*,section+)>
<!ELEMENT section     (title,body*,subsection+)>
<!ELEMENT subsection  (title,body+)>
<!ELEMENT body        (paragr|figure|table)>
<!ATTLIST body         mdate  #CDATA>
 ...
```

(a) DTD 정의



(b) OODB 스키마

그림 1: XML DTD와 변환된 OODB 스키마

```
where b.*.body.mdate = '2000:01:01';
```

여기서 '*' 문자를 포함하는 b.*.body.mdate와 같은 경로식을 '확장 경로식(extended path expression)'이라고 정의하며 XML DTD 또는 변환된 OODB 스키마에서 가능한 모든 스키마 경로를 나타낸다. 따라서 b.*.body.mdate는 b.chapter.body.mdate, b.chapter.section.body. mdate, b.chapter.section.subsection.body.mdate를 동시에 나타내는 것이다. 앞서 언급한 기존의 인덱스 기법은 모두 스키마 상의 단일 경로에 대한 객체 탐색을 지원하는 인덱스 기법이므로 본 논문에서 제안하는 기법과 구분하여 '단일 경로 인덱스'로 정의한다.

다음은 기존의 단일 경로 인덱스를 사용하여 확장 경로식에 대한 인덱스 기능을 지원하는 경우 나타날 수 있는 문제점들이다.

- 확장된 경로에 해당하는 개개의 경로의 개수가 많아지면 전체 인덱스의 개수가 많아지게 된다. 따라서 인덱스 저장 공간이 그에 비례하여 늘어나고 검색 성능도 떨어지게 된다. 특히 일치 검색(exact match key look-up)의 경우에는 성능이 급격히 저하된다. 또한 영역 검색(key range look-up) 시에는 개개의 인덱스를 검색한 결과를 병합해야 하는 경우가 있어 병합에 따르는 부가적인 오버헤드가 발생한다.

- 인덱스에 대한 통합적인 관리가 어렵다. 즉 확장 경로에 해당하는 개개의 단일 경로를 데이터베이스 관리자가 알아내어 각각에 대해 인덱스를 설정해야 한다. 또한 인덱스를 삭제할 때에도 관련된 확장 경로에 대해 설정된 인덱스가 어떤 것들이 있는지를 관리자가 파악해야 한다. 이 과정에서 확장된 경로에 포함되는 일부 단일 경로에 대해서는 인덱스가 존재하나 나머지 경로에 대해서는 그렇지 않을 수도 있게 된다. 즉 일관성 있는 인덱스 관리가 어렵고 인덱스를 이용하는 이득이 감소하게 된다.

이에 본 논문에서는 경로의 개수가 많아져도 우수한 성능을 보이며 관리하기에도 편리한 확장 경로에 대한 인덱스 기법으로서 다중 경로 인덱스 기법(multi-path index scheme)을 제안하고자 한다.

본 논문은 다음과 같이 구성된다. 이어지는 장에서는 경로에 대한 인덱스 기법과 관련된 기존 연구를 조사하였다. 3 장에서는 제안하는 인덱스 기법을 사용하기 위한 조건 및 본 논문에서 사용하는 몇 가지 용어들에 대한 정의를 설명한다. 4 장에서는 제안하는 인덱스의 구조 및 구성 방법을 설명한다. 5 장은 제안하는 기법의 실험을 통해 기존 방식과의 차이점을 살펴보고 그 특성을 알아본다. 마지막으로 6 장은 결론을 맺는다.

2 관련 연구

XML 도입 초기 단계에는 XML 데이터를 주로 파일 시스템을 이용하여 관리하였으나 데이터의 양이 급증하고서부터는 데이터베이스 시스템을 이용하는 것이 일반적인 추세이다. 그에 따라 RDB나 OODB를 이용하여 XML(또는 SGML) 데이터를 관리하기 위한 연구가 수행되었다. [2, 8, 9] 는 RDB를 이용하여 XML 데이터를 관리하는 기법을 제안하고 있다. 이들 기법에 따르면 RDB의 장점인 시스템의 안정성, 확장성, 최적화된 질의 처리 등을 활용하는 면에서 유리하다. 반면 OODB를 이용한 방식은 데이터베이스의 데이터 모델과 XML의 모델이 거의 유사하므로 XML 데이터의 저장과 검색이 자연스럽게 이루어질 수 있다는 장점이 있다. [3, 4]는 OODB를 이용한 방식에 대해 다루고 있다.

OODB를 이용하는 방식은 XML의 요소를 주로 객체 단위로 저장하기 때문에 서로 참조 관계에 있는 객체들을 효율적으로 탐색하는 것이 검색 성능과 관련하여 중요한 관건이 된다. 이와 같이 경로 상에 존재하는 객체들의 탐색을 효과적으로 지원하기 위해 중첩 인덱스와 경로 인덱스, 다중 인덱스, ASR, Object Skeleton [10] 등의 기법들이 제안되었다. 이들은 질의의 형태, 변경 연산의 비율, 저장 공간의 소모량 등의 측면에서 각각 장단점을 가지고 있지만 공통적으로 단일 스키마 경로에 대한 객체 탐색을 지원하기 위한 인덱스 기법들이라는 점이다. 따라서 앞장에서와 같이 '*' 를 포함하는 확장 경로식에 대한 인덱스 기능을 제공하기에는 곤란하다.

유니폼(uniform) 인덱스 기법 [11]은 경로 상의 모든 클래스와 그것들의 하위 클래스에 대해 각각 고유한 경로 식별자를 할당하고 경로 인덱스의 단말 노드에 해당 경로가 통과하는 클래스들의 경

로 식별자를 기록하도록 한 것이다. 즉 OODB의 특징과 관련된 두 가지 인덱스 기능인 클래스 계층 인덱스 기능과 경로 인덱스 기능을 동시에 지원하기 위한 인덱스 기법이다.

이 기법은 본 논문이 제안하는 방법과 마찬가지로 비교적 간단한 방법으로 클래스 계층 인덱스와 경로 인덱스를 동시에 지원할 수 있고 B^+-트리를 크게 변형하지 않고 사용할 수 있으므로 실용성도 높다. 반면 [11]의 기법은 제안하는 기법과는 달리 단일 인덱스 경로에 위치하는 상위-하위 클래스들을 구분하여 검색하기 위한 기법이라는 제한점이 있다. 즉 논리적으로는 단일 경로에 대한 인덱스 기법이다. [12]에서는 1-index, 2-index, 그리고 T-index를 제안하고 있다. 이들은 주어진 경로식에 대한 오토마타와 오토마타의 각 노드에 해당하는 경로에 대한 인덱스 정보를 데이터베이스에 저장하는 기법으로 볼 수 있다. 즉, 정규식으로 표현된 개개의 경로에 대한 ASR [13]을 익스텐트(extent)로 각각 구성하고 오토마타를 통해 원하는 경로에 대한 ASR을 선택하는 기법이다. 이때 ASR은 각 경로 상의 모든 객체들의 객체 식별자(object identifier, OID)를 경로 인덱스 [6]로 유지하는 것이 아니라 경로 상의 지정된 몇몇 클래스에 속하는 객체만을 유지한다.[1] 그러나 Milo 인덱스 기법은 동일한 인덱스 정보가 반복해서 저장되기 때문에 저장공간을 많이 사용하고 인덱스 갱신에 따른 오버헤드가 크다. 따라서 다중 경로에 대한 검색시 단일 경로 인덱스 기법과 마찬가지로 경로의 수가 많아질수록 검색에 따른 오버헤드가 증가하게 된다.

DataGuide [14]는 XML과 같은 특징을 갖는 반 구조화된 데이터(semistructured data)에 대한 저장 시스템인 LORE에서 효과적으로 사용될 수 있는 인덱스 구조이다. 그러나 DataGuide는 특정 경로에 해당하는 모든 객체들의 객체 식별자를 모아 놓은 일종의 클래스 익스텐트 객체와 유사하므로 경로 인덱스로 보기는 어렵다.

3 제약 조건 및 용어 정의

이 장은 제안하는 다중 경로 인덱스 기법을 사용할 수 있는 환경에 대한 조건과 본 논문에서 사용하는 용어들에 대한 정의를 설명한다.

3.1 다중 경로 인덱스를 사용할 수 있는 조건

제안하는 다중 경로 인덱스를 사용하게 될 XML DTD의 특성과 관련하여 다음과 같은 조건을 가정한다.

먼저 XML DTD 내의 요소(element)와 속성(attribute)의 이름은 해당 DTD 내에서는 유일해야 하며 다른 DTD에 동일한 이름을 가진 것이 있더라도 서로 다른 것으로 한다. 마찬가지로 요소에 대응하여 생성되는 OODB에서의 클래스도 서로 다른 XML DTD에 존재하는 동일한 이름의 요소인 경우에는 서로 다른 것으로 가정한다. 그리고 동일한 DTD 내의 요소와 속성의 이름 중 서로 동일한

[1] 1-index의 경우 단말 노드에 속하는 객체, 2-index의 경우에는 시작과 끝 객체, T-index의 경우 지정된 객체들.

것이 없어야 한다. 그리고 제안하는 기법은 DTD 수준에서 순환(cycle)이 존재하는 경로에 대해서는 인덱스를 설정할 수 없다. 이 기능은 차후 개선될 예정이다. 또한 동일한 이름의 속성이 다수의 요소에 대해 사용될 경우에는 그것들의 데이터 타입이 모두 동일한 것으로 가정한다.

제안하는 기법은 XML의 IDREF 형의 속성에 대한 인덱스 경로는 지원하지 않는다. 즉 IDREF를 통해 다른 XML 객체를 가리키는 경우나 동일한 XML 객체 내의 다른 객체를 가리키는 경우들에 대해서는 질의 경로를 사용할 수 없는 것으로 한다. 이것은 IDREF의 경우 실제 그것이 참조할 대상이 어떤 타입의 객체가 될 지 알 수 없으므로 OODB의 스키마 상에서 참조관계를 그래프로 표현할 수 없게 되고 따라서 경로 인덱스를 사용할 수가 없기 때문이다.

3.2 용어 정의

객체 질의어에서 사용되는 기존의 경로식은 하나의 경로만을 나타내므로 단일 경로식이라 할 때, 기존 경로식 표현에 '*' 문자를 결합하여 다음과 같은 형태로 나타낸 경로식을 완화 경로식(relaxed path expression)이라 하자.

$$P_f.*.P_r$$

여기서 P_f와 P_r은 '*'를 포함하지 않는 단일 경로식으로 각각 전경로(pre_path)와 후경로(post_path)로 정의한다. XML의 요소는 OODB의 스키마로 사상된 후에는 클래스에 해당하기 때문에 본 논문에서는 요소와 클래스를 동등한 의미로 사용한다. 그리고 기존의 경로식과 완화 경로식을 합쳐 확장 경로식으로 정의한다. 따라서 `A.B.C.D` 와 `A.B.C.D.D1`과 같은 기존의 경로식은 단순 경로식이고 `A.*.D` 또는 `A.B.*.D.D1`과 같은 경로식은 완화 경로식으로 볼 수 있으며 네 가지 모두 확장 경로식으로 볼 수 있다. `A.*.D`와 `A.B.*.D.D1`의 전경로는 각각 `A`와 `A.B`이고 후경로는 각각 `D`와 `D.D1`이 된다.

XML의 DTD 요소 또는 OODB의 스키마에서 나타난 클래스의 이름에 의한 경로식을 '스키마 경로'라 하고, 그것이 질의에 사용되었을 때는 '질의 경로'라 정의한다. 스키마 경로에 따라 데이터베이스에 저장된 일련의 객체들은 '경로 인스턴스(path instance)' 또는 '객체 경로(object path)'로 정의한다.

4 다중 경로 인덱스 기법

이 장에서는 다중 경로 인덱스의 구현 방법에 대해 설명한다. 임의의 확장 경로식이 주어졌을 때, 다중 경로 인덱스는 다음과 같은 과정으로 구현된다.

1. 다중 경로 인덱스를 구현하기 위해서는 먼저 주어진 확장 경로식에 해당하는 개개의 단일 경로를 찾아내고 각각에 대해 '경로 식별자(path identifier, PID)'를 할당한다.

2. 질의 처리기를 통해 단일 경로별로 데이터베이스 내의 실제 데이터를 검색하여 인덱스 정보를 수집한다.

3. 수집된 인덱스 정보를 다중 경로 인덱스에 저장한다.

여기서 두 번째 과정은 질의 처리기를 이용하여 처리되므로 그 과정에 대한 자세한 처리 과정은 생략하고 (1) 경로 식별자를 할당하는 방법과 (2) 다중 경로 인덱스에 저장되는 정보 및 저장 구조 (3) 다중 경로 인덱스에 대한 검색 방법 및 갱신 방법을 차례로 설명한다.

4.1 경로 식별자의 할당

XML DTD와 인덱스를 설정할 확장 경로식 P가 주어졌을 때, 주어진 XML DTD(또는 변환된 OODB 스키마)를 만족하면서 P에 속하는 개개의 단일 경로식을 해당 경로식에 대한 '멤버 경로식' 또는 '멤버 경로'로 정의한다. 그리고 P의 모든 멤버 경로식의 집합을 '경로 익스텐트'라 정의하고 $\varepsilon(P)$로 나타낸다. 알고리즘 1은 주어진 확장 경로식에 대한 경로 익스텐트를 구하고 경로 식별자를 할당하는 알고리즘을 나타낸 것이다. XML DTD를 OODB 스키마로 변환하였을 때, 질의문에서 경로식으로 표현될 수 있는 곳은 원래의 XML DTD에서 부모 요소 - 자식 요소 또는 요소 - 속성 관계이므로 이 점을 이용하여 경로 익스텐트를 찾아낼 수가 있다. 먼저 알고리즘 1에서 사용되는 $head(P)$와 $tail(P)$는 다음과 같이 정의한다.

- $head(P)$: 경로 P의 가장 앞쪽 요소
- $tail(P)$: 경로 P의 가장 뒤쪽 요소 혹은 속성

그리고 요소 E에 대한 $child(E)$는 E를 구성하는 자식 요소 또는 E의 속성들의 집합을 나타내는 것으로 한다.

알고리즘 1은 주어진 경로식이 '*' 를 포함하지 않는 단일 경로식인 경우에는 주어진 경로식만으로 경로 익스텐트를 삼는다. 만일 완화 경로식이 주어진 경우에는 $tail(pre_path(P))$에 해당하는 요소로부터 $head(post_path(P))$에 해당하는 모든 멤버 경로를 찾기 위해 search_path 함수를 호출한다. search_path 함수는 인자로 넘겨받은 요소(e)의 자식(c) 중에 $head(post_path(P))$에 해당하는 것이 있는지를 검사한다. 만일 조건에 맞는 자식을 찾았으면 스택에 있는 현재까지의 탐색 경로 정보와 해당 자식 요소(혹은 속성)를 합쳐서 경로 익스텐트 집합에 삽입한다. 만일 c가 부모 요소이면 search_path 함수에 대한 재귀 호출(recursive call)을 통해 탐색을 계속한다. 이러한 탐색 과정은 자식이 없는 단말 요소(leaf element)나 속성을 방문할 때까지 계속된다. 익스텐트가 구해지고 나면 찾아진 각각의 단일 경로에 대해 경로 식별자를 할당한다.

예제 1 *그림 1-(a)의 XML DTD에 대해 알고리즘 1을 이용하여 확장 경로식* `book.chapter.*.title`*의 경로 익스텐트에 속하는 멤버 경로에 경로 식별자를 할당하면 다음과 같다.*

Algorithm 1 경로 식별자를 할당하는 알고리즘

```
1: Input: XML DTD, p : extended path expression
2: Output: ε(p)
3:
4: if p is a single path expression then
5:    ε(p) ← p;
6:    return
7: end if
8: ε(p) ← ∅
9: P_f ← pre_path(p)
10: P_r ← post_path(p)
11: check validity of path P_f and P_r
12: from ← tail(P_f)
13: to ← head(P_r)
14: initialize stack
15: search_path(from)
16: assign a serial number for each member path in ε(P)
17:
18: procedure search_path(e)
19: if e is leaf element then
20:    end procedure
21: end if
22: push e into stack
23: for all c ∈ child(e) do
24:    if c = to then
25:       ε(p) ← ε(p) ∪ { concatenate (path in stack, c) }
26:    else if c is element then
27:       search_path (c)
28:    end if
29: end for
30: pop stack
31: end procedure
```

```
book.chapter.title - PID : 1
book.chapter.section.title - PID : 2
book.chapter.section.subsection.title - PID : 3
```

4.2 다중 경로 인덱스에 유지되는 정보 및 저장 구조

4.2.1 다중 경로 인덱스에 유지되는 정보

확장 경로식 P 에 대한 다중 경로 인덱스 i는 다음과 같은 정보들을 유지하는데 이러한 정보는 인덱스에 대한 검색과 변경 및 인덱스의 선택에 사용된다.

- P_i – 인덱스 경로
- D_i – 인덱스의 방향, 즉 순방향(forward, 경로의 방향과 일치) 또는 역방향 (backward, 경로의 역방향)
- H_i – 경로의 시작 클래스로서 순방향 인덱스의 경우에는 $head(pre\_path(P))$, 역방향의 경우에는 $tail(post_path(P))$에 해당한다.
- T_i – 경로의 끝 클래스로서 순방향 인덱스의 경우에는 $tail(post\_path(P))$, 역방향의 경우에는 $head(pre_path(P))$에 해당한다.
- E_i – 인덱스 i를 통해 검색 가능한 모든 멤버 경로 $= \varepsilon(P)$
- BT_i – 인덱스 데이터를 저장하고 있는 B^+-트리 객체의 객체 식별자
- 각 멤버 경로별 정보 : $sp \in \varepsilon(P)$
  - sp – 멤버 경로의 경로식
  - $\eta(sp)$ – 경로에 해당하는 경로 인스턴스의 개수
  - PID – 경로 식별자

4.2.2 다중 경로 인덱스의 저장 구조

다중 경로 인덱스의 인덱스 정보는 단말 레코드의 구조를 변경시킨 B^+-트리를 이용하여 저장한다. 그림 2는 다중 경로 인덱스의 단말 레코드의 구조를 나타낸 것이다. 그림의 각 부분이 의미하는 바는 다음과 같다.

- record header – 해당 레코드의 크기 등을 기록한 헤더 정보
- key value – 인덱스 키값

record header	key value	# paths	(PID_1, #OIDs, OIDs)	$\cdots$	(PID_n, #OIDs, OIDs)

그림 2: 다중 경로 인덱스 단말 노드의 레코드 구조

- # paths – 서로 다른 경로의 개수
- PID – 해당 경로의 경로 식별자
- $\#OIDs$ – 경로에 해당하는 객체 식별자의 개수
- $OIDs$ – 경로에 해당하는 객체의 객체 식별자들

여기서 경로 식별자는 앞 절에서 설명된 경로 식별자 할당 알고리즘에 의해 주어진다. 다중 경로 인덱스의 중간 노드 구조는 일반적인 B^+-트리의 중간 노드 구조와 동일하다. 다중 경로 인덱스는 인덱스 키값과 해당되는 키값에 연결된 경로의 다른 쪽 끝에 있는 객체의 객체 식별자를 저장하는 면에서 [6]에서 제시된 중첩 인덱스(nested index)와 유사하다.

예제 2 *그림 3은 그림 1에서 제시된 XML DTD 스키마에 따르는 XML 문서들을 OODB에 저장한 것으로 데이터베이스의 일부만을 나타낸 것이다. 그림 3에서 원은 객체(object)를 의미하며 원 내의 숫자는 해당 객체의 객체 식별자를 의미한다. 원 옆의 숫자는 mdate의 값이다. 'Books'로 표시된 o0 객체는 book 클래스의 모든 객체에 대한 객체 식별자를 유지하고 있는 book 클래스의 익스텐트 객체이고 o1과 o2는 각각 book 타입의 XML 객체를 저장하는 루트 요소에 해당하는 객체이다. 다음은 주어진 데이터베이스에 대해 경로 book.chapter.*.mdate에 대한 다중 경로 인덱스를 i라 할 때 그것의 구조와, 동일한 경로에 대해 기존의 중첩 인덱스를 이용했을 때 그것의 단말 레코드 구조를 보인 것이다.*

– **다중 경로 인덱스**

- P_i – *book.chapter.*.mdate*
- D_i – *backward*
- H_i – *book*
- T_i – *mdate*
- E_i – *book.chapter.body.mdate,*
  book.chapter.section.body.mdate,
  book.chapter.section.subsection.body.mdate
- *멤버 경로별 정보*

- *book.chapter.body.mate – PID: 1, η: 2*
- *book.chapter.section.body.mdate – PID : 2, η : 2*
- *book.chapter.section.subsection.body.mdate – PID : 3, η : 7*

- *인덱스 내의 단말 레코드의 생성 과정 – 각 멤버 경로별로 키값과 포인터 값의 쌍을 추출한 다음 다중 경로에 삽입한다*[2]

  1. *PID 1 에 해당하는 멤버 경로의 키값과 포인터 값의 쌍을 추출한 결과:*
     ('2000:01:01', o1), ('1999:11:03', o2)
  2. *위에서 추출된 결과를 다중 경로 인덱스에 삽입한 후에 생성되는 다중 경로 인덱스의 단말 레코드들:*
     ('1999:11:03', 1,(1, 1, o2)),
     ('2000:01:01', 1,(1, 1, o1))
  3. *PID 2에 해당하는 멤버에 대해 위의 PID 1에서와 같은 과정을 거친 후의 다중 경로인덱스 단말 레코드들:*
     ('1999:11:03', 2, (1, 1, o2) (2, 1, o1)),
     ('2000:01:01', 2, (1, 1, o1) (2, 1, o2))
  4. *PID 3에 해당하는 경로에 대해 위에서와 같은 과정을 거친 이후의 다중 경로인덱스 단말 레코드들(최종):*
     ('1999:11:03', 3, (1, 1, o2) (2, 1, o1) (3, 1, o1)),
     ('2000:01:01', 3, (1, 1, o1) (2, 1, o2) (3, 4, o1 o1 o2 o2)),
     ('2000:02:01', 1, (3, 2, o1 o2))

– 중첩 인덱스

중첩 인덱스의 단말 노드 구조는 그림 2의 인덱스 단말 노드 구조에서 #paths 와 $PID_k(k = 1, \ldots, N)$ 및 $\#OIDs$ 필드를 제외한 것과 같은 구조이다. 그리고 단일 경로 인덱스이므로 확장 경로의 각 멤버 경로에 대해 하나의 인덱스가 생성된다. 각 경로별로 생성된 인덱스의 단말 레코드를 나타내면 다음과 같다.

- *book.chapter.body.mdate에 대한 중첩 인덱스*
  ('1999:11:03', o2), ('2000:01:01', o1)

[2] 다중 경로 단말 레코드의 헤더 정보는 생략하여 나타내었다.

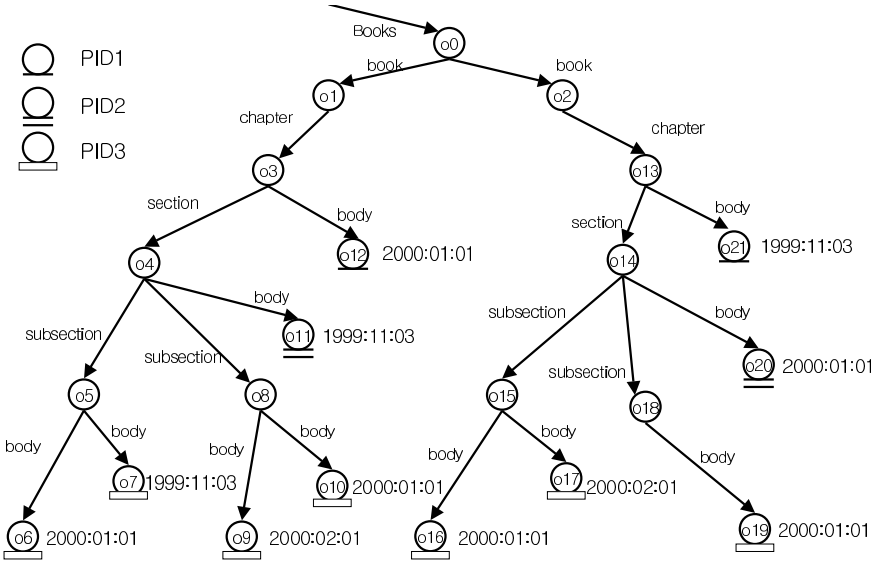


그림 3: XML 문서를 OODB에 저장한 예

- *book.chapter.section.body.mdate 에 대한 중첩 인덱스*
  ('1999:11:03', o1), ('2000:01:01', o2)
- *book.chapter.section.subsection.body.mdate 에 대한 중첩 인덱스*
  ('1999:11:03', o1), ('2000:01:01', o1 o1 o2 o2), ('2000:02:01', o1 o2)

4.3 다중 경로 인덱스의 검색 및 갱신 방법

다중 경로 인덱스에 대한 검색 과정은 일반 B^+-트리와 동일하되 단말 레코드의 검색만을 달리한다. 즉 B^+-트리와 동일한 방식으로 인덱스 트리를 탐색하되 최종적으로 단말 노드에서만 주어진 질의 경로를 만족하는 객체 식별자만을 선택하면 된다. 따라서 원하는 검색 경로에 대한 경로 익스텐트를 모두 구하고 경로 익스텐트 내의 각각의 단일 경로에 대한 경로 식별자를 찾아낸다. 그리고 나면 주어진 키값으로 중간 노드와 최종 단말 노드를 찾은 다음, 키값과 경로 식별자의 값이 일치하는 객체 식별자만을 검색하면 된다. 알고리즘 2은 이 과정을 나타낸 것이며 다음은 이 과정에 대한 예이다.

예제 3 *예제 2에서 사용한 다중 경로 인덱스와 그림 3의 데이터베이스를 이용하여 다음의 두 질의를 처리하는 과정이다.*

1. `select b from b in Books where b.chapter.section.*.mdate = '1999:11:03';`
2. `select b from b in Books where b.chapter.*.mdate = '2000:02:01';`

첫 번째 질의의 질의 경로인 book.chapter.section..mdate의 경로 익스텐트는 book.chapter.section.body.mdate와 book.chapter.section.subsection.body.mdate이고 해당되는 PID는 2와 3이 된다. 따라서 주어진 mdate*

Algorithm 2 다중 인덱스의 검색 알고리즘

```
1: Input: key(key value), pids(set of PIDs)
2:
3: traverse index nodes using the key in the same manner with B+ tree.
4: if key not found then
5:    return
6: end if
7:
8: r ← leaf record having the key
9: s ← ∅
10: for all PID in r do
11:    if PID exists in pids then
12:       s ← s ∪ OIDs corresponding to this PID
13:    end if
14: end for
15: return s
```

값인 '1999:11:03'으로 인덱스를 검색한 다음, 단말 노드 중에서 키값이 일치하고 PID가 2 또는 3이 되는 객체 식별자를 검색하면 book 클래스의 객체 o1을 찾을 수 있다. 두 번째 질의의 경우에는 질의 경로에 대해 예제 2의 인덱스 경로가 모두 해당되므로 PID 값에 상관없이 키값 '2000:02:01'과 일치하는 것을 모두 찾으면 book 클래스의 객체 o1, o2가 검색된다.

알고리즘 3와 알고리즘 4은 다중 인덱스에 대한 삽입과 삭제 알고리즘을 설명한 것이다. 인덱스에 대한 갱신은 삭제 알고리즘과 삽입 알고리즘을 이용하여 구현한다.

Algorithm 3 다중 인덱스의 삽입 알고리즘

```
1: Input: key(key value), objid(OID), pid(PID)
2: traverse index nodes using key
3: if key found then
4:    append objid in OID list of corresponding PID
5: else
6:    allocate a new leaf record, and insert key, pid, objid
7: end if
8: adjust tree strucutr in the same manner as in B+ tree.
```

Algorithm 4 다중 인덱스의 삭제 알고리즘

```
1: Input: key(key value), objid(OID), pid(PID)
2: traverse index nodes using key
3: r ← the leaf record found
4: remove (pid, objid) pair from r
5: if given pid has no OID then
6:    remove corresponding (PID, OID) pairs
7: end if
8: if given r has no OID then
9:    remove r from the index
10: end if
11: adjust tree strucutr in the same manner as in B+ tree.
```

다중 경로 인덱스는 중첩 인덱스와 같이 지정된 경로에 대해 키값과 경로의 끝에 해당하는 객체의 객체 식별자를 유지하는 방식이다. 따라서 객체들간의 참조 관계가 삽입, 삭제 또는 변경되는 경우에는 중첩 인덱스와 동일하게 삽입 또는 삭제가 일어난 객체 경로의 양끝을 탐색하여 경로에 대한 변경 정보를 구하고 그것을 인덱스에 반영하여야 한다. 이 과정에 대해서는 참고 문헌 [6]에 나타나 있다.

4.4 순수 경로 인덱스

경로 인덱스 중에서 인덱스의 키값으로서 객체 식별자를 사용하는 경우에 해당 인덱스 객체를 '일치 인덱스(identity index)' [15]라고 한다. 객체 식별자 인덱스는 경로 인스턴스의 한쪽 끝에 해당하는 객체의 객체 식별자를 키값으로 하고 다른 한쪽 객체의 객체 식별자를 인덱스의 포인터 값으로 사용하는 경우이다. 일치 인덱스는 주로 다음 질의와 같이 인덱스 검색 조건이 요소 자체 문자열에 대한 검색 조건을 포함하는 경우에 사용된다.

예제 4 *서적들 중에서 본문의 내용 가운데 'XML'이라는 문자열을 포함하고 있는 것을 검색하라.*

```
select b from b in Books where contains(b.*.paragr, 'XML');
```

paragr 객체는 서적의 본문을 나타내는 것으로서 이것은 일반적으로 수십 ~ 수백 개 이상의 단어로 구성되는 텍스트(text)이다. 이 경우, paragr 객체는 그 크기와 예상되는 검색 형태가 인덱스의 키값으로 사용하기에는 부적합하다. 따라서 이 때는 paragr 클래스에 속하는 객체들의 객체 식별자와 해당 객체와 관련된 Book 객체의 객체 식별자를 쌍으로 하는 객체 식별자 인덱스를 구성한다. 그리고 나면 위와 같은 질의가 주어진 경우 다음과 같은 과정으로 질의를 처리할 수 있다. 먼저 질의 처리기는 paragr 클래스에 속하는 객체를 모두 검색하여 'XML' 문자열을 포함하는 객체들을 찾는다. 그리고 추출된 객체들의 객체 식별자로부터 참조 관계로 연결된 Book 객체를 찾는다. 이 때 경로 Books.*.paragr에 대해 설정된 일치 인덱스를 사용하면 검색 성능을 향상시킬 수가 있다.

본 논문에서는 일치 인덱스가 키값을 갖는 객체의 속성 값이나 텍스트와 같은 '값'에 기반을 두지 않고 순수하게 객체들간의 특정 경로에 따라 참조 관계에 있는 객체들 간의 참조 관계만을 유지하므로 '순수 경로 인덱스 (pure path index)'로 정의한다. 이와는 반대로 객체 식별자가 아닌 개체의 속성 값이나 텍스트를 키로 사용하는 인덱스를 '값 인덱스(value index)'로 구분하겠다. 여기서 주목해야 할 점은 순수 경로 인덱스는 기존의 단일 경로 인덱스 방식이나 제안하는 다중 경로 인덱스 방식에서 모두 사용될 수 있는 것으로 단지 키값의 특성에 따라 구분한 것이며 인덱스 구조와는 관계가 없다는 것이다. 그리고 키값으로서 객체 식별자를 사용하는 경우이므로 주어진 단일한 키값에 해당하는 객체들을 검색하기 위한 일치 검색 연산(exact match key look-up)만이 의미가 있으며 일정 범위의 키값에 해당하는 객체들을 검색하기 위한 영역 검색 연산(key range look-up)은 의미가 없다는 점이다.

5 성능 평가

이 장에서는 임의의 확장 경로식에 대해 다중 경로 인덱스를 사용한 인덱스 방법과 여러 개의 단일 경로 인덱스를 사용한 방식과의 성능을 비교하였다.

성능 비교에 사용할 단일 경로 인덱스 방법으로는 중첩 인덱스 [6]를 사용하였다. 중첩 인덱스를 사용한 이유는 제안하는 다중 경로 인덱스 방법과 동일하게 키 값과 경로의 끝에 해당하는 객체의 객체 식별자의 쌍을 저장하는 방식을 사용한다는 점이다. 또한 중첩 인덱스 방식이 인덱스가 설정된 경로에 대해서는 다른 인덱스 방식보다 우수한 성능을 보인다는 점이다 [6]. 이제부터는 다중 경로 인덱스 방식을 이용한 경우를 MPI, 중첩 인덱스를 사용한 경우를 GNI로 나타낸다.

5.1 실험 환경

그림 4는 실험에 사용된 경로식을 OODB의 스키마 형태로 나타낸 것이다. 그림에서 C_t는 스키마 경로의 시작 클래스이며 C_p는 스키마 경로의 끝 클래스이고 두 클래스 사이에는 N개의 서로 다른 멤버 경로가 존재한다. $kattr$는 인덱스의 키로 사용되는 C_p의 속성이다. 즉, 본 실험은 확장 경로 $C_t.*.C_p.kattr$에 대한 인덱스 성능 평가라 할 수 있다.

표 1은 실험에 사용된 인자들을 나타낸다. 중첩 인덱스의 경우에는 각 경로에 대해 인덱스를 하나씩 설치하게 되므로 경로의 개수만큼의 인덱스를 사용하게 된다. 반면 다중 경로 인덱스의 경우에는 경로의 개수에 상관없이 하나의 인덱스만을 사용한다. c는 C_t 클래스에 속하는 객체들의 개수이다. f는 경로당 팬-아웃으로서 각 멤버 경로에 대해 C_t의 객체로부터 연결되는 클래스 C_p의 객체의 개수, 즉 경로 인스턴스의 개수를 나타낸다. 따라서 $N = 8$이고 $f = 10$일 때는 C_t의 한 객체에 대해 8개의 서로 다른 멤버 경로가 존재하고 각 멤버 경로에 대해 10개씩의 객체가 연결되어 있음을 의미한다. 따라서 C_t 객체 하나에 모두 80개의 C_p 객체가 연결된다. 따라서 실험에서와 같이 C_t 클래스에 속하는 객체가 1000개이면 그것들에 연결된 C_p 객체는 모두 80,000 개가 된다. kl은 키값 kattr의 크기이고 kr은 키값의 중복도를 나타내는 것으로서 동일한 키값을 갖는 C_p 객체의 개수를 의미한다. rqr는 키값의 전체 범위에 대한 질의 영역의 범위를 나타낸 것이다.

실험 인자	값
c	1000
kl	8, 64
N	2, 4, 6, 8, 10
f	1, 5, 10
kr	1, 10, 100
rqr	1%, 5%

표 1: 실험 인자

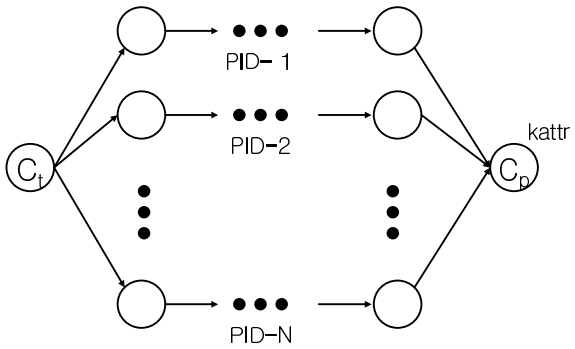


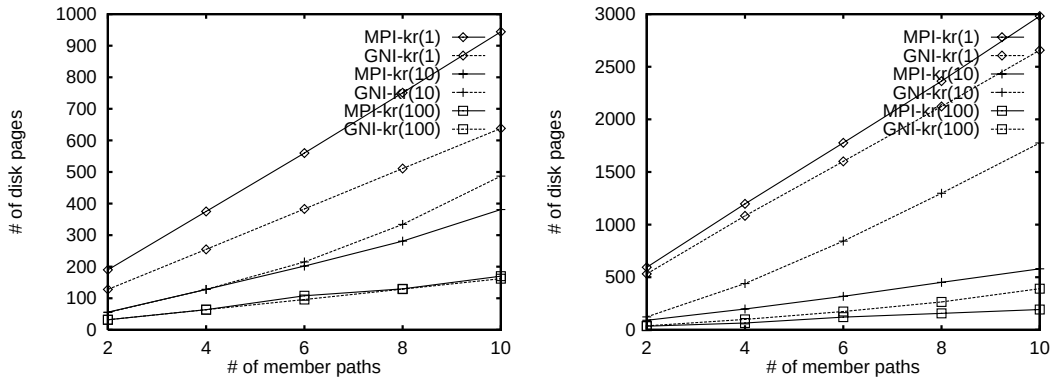
그림 4: 실험에 사용된 OODB 스키마 구조

실험은 각 경우에 대해 MPI와 GNI의 저장 공간 비용(storage cost)과 검색 성능(look-up performance)을 비교하였다. 저장 공간 비용은 각각의 인덱스 방식을 적용한 경우 소모되는 디스크 페이지의 양을 비교한 것이다. 검색 성능 비교는 다시 일치 질의 검색 연산 성능과 영역 검색 연산 성능으로 구분하여 비교하였다. 일치 검색 연산은 주어진 키값에 정확히 일치하는(exact match) 키값만을 검색하는 것이고 영역 검색 연산은 일정 범위의 키값에 해당하는 객체를 검색하는 것이다. 검색 성능 비교는 각 인덱스 기법에 대해 단일한 검색 연산을 수행하는데 소요된 시간을 기준으로 하였다.

MPI와 GNI는 다음과 같이 구성된다. GNI의 경우에는 클래스 C_t로부터 속성 $C_p.kattr$에 도달할 수 있는 모든 가능한 스키마 경로인 N 개의 스키마 경로에 대해 각각 중첩 인덱스를 사용한다. 따라서 GNI의 경우에는 모두 N 개의 인덱스를 사용하게 된다. MPI의 경우에는 알고리즘 1을 이용하여 각각의 경로에 대해 그림 4와 같이 $PID = 1, \ldots, PID = N$을 할당한 뒤 하나의 인덱스에 저장한다. 모든 인덱스는 서로 독립적인 디스크 세그먼트(연속된 페이지들의 집합)에 저장함으로써 개개의 인덱스에 속하는 노드들이 서로 클러스터링(clustering) 되도록 하였다.

5.2 실험 결과

그림 5는 실험 결과 생성된 인덱스들이 차지하는 디스크 공간을 키의 크기에 따라 나타낸 것이다. $kr = 1$인 경우에는 MPI 인덱스 방식이 GNI 방식보다 많은 저장 공간을 차지하는데 그것은 MPI의 단말 노드 레코드가 GNI 방식의 단말 노드 레코드보다 크기 때문에 더 많은 수의 인덱스 단말 노드를 사용하기 때문이다. 그러나 이와 같은 저장공간의 차이는 키의 크기가 클수록($kl = 64$) 줄어드는데 그것은 키의 크기가 큰 경우에는 MPI의 단말 노드 레코드와 GNI의 단말 레코드의 크기의 차가 상대적으로 줄어들기 때문이다. $kr = 10$ 또는 100의 경우처럼 중복되는 키값이 많은 경우일수록 MPI 방식이 더 작은 저장공간을 사용하게 되는데 그것은 MPI의 경우 단말 레코드 하나에 더 많은 수의 포인터(객체 식별자)를 저장하기 때문에 결과적으로 더 적은 수의 단말 노드를 사용하게 되기 때문이다. 이러한 경향은 그림 5-(b)에서 보듯이 키값의 크기가 큰 경우에 더욱 강하게 나타나게 된



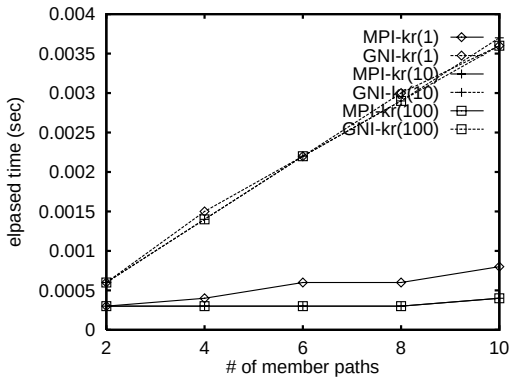
(a) kl = 8 (b) kl = 64

그림 5: 저장 공간 사용량

다. 종합적으로는 인덱스 키값이 큰 경우 또는 중복되는 키값의 개수가 많아질수록 MPI가 저장 공간 측면에서는 유리하게 됨을 알 수 있다.

그림 6은 $kl = 8$, $f = 10$의 경우에 대한 일치 검색 질의 성능 평가 결과를 나타낸 것이다. 실험 인자를 달리한 경우에도 유사한 결과를 보이므로 생략하였다. 그림 6은 MPI가 모든 키값의 중복도 여부에 상관없이 항상 더 좋은 성능을 보이며 특히 멤버 경로의 개수가 많아질수록 성능의 차이가 커지는 것을 알 수 있다. 이러한 결과는 일치 검색의 경우 인덱스 높이에 영향을 받기 때문이다. 즉 GNI의 경우 멤버 경로에 해당하는 각각의 인덱스에 대해 모두 일치 검색을 수행해야 하므로 '멤버 경로 인덱스의 개수(경로의 개수) × 멤버 경로 인덱스의 높이'에 해당하는 인덱스 페이지를 방문해야 하는 반면, MPI의 경우 인덱스가 하나만 존재하므로 항상 인덱스의 높이에 해당하는 개수의 인덱스 페이지만을 방문하면 되기 때문이다. 위와 같은 사실은 그림 6에 나타난 그래프들이 거의 유사하게 변화하는 현상으로부터 더욱 확실히 알 수 있다.

그림 7은 영역 검색 실험 결과들 중의 일부를 보인 것이다. $kl = 8$, $rqr = 5\%$의 경우를 제외하고는 MPI가 우수한 성능을 보인다. 특히 전반적으로 키값의 중복도가 클수록 MPI가 좋은 성능을 나타낸다. 이것은 키의 중복도가 커질수록 MPI는 GNI 보다 더 적은 개수의 단말 노드를 생성하므로 영역 질의를 처리하는 과정에서 탐색해야하는 단말 노드의 개수가 줄어들기 때문이다. 키의 중복이 없는 $kr = 1$의 경우는 MPI가 GNI보다 많은 수의 단말 노드를 사용함에도 불구하고 $kl = 8, rqr = 5\%$의 경우를 제외하고는 MPI가 더 우수한 성능을 나타낸다. 이러한 현상은 MPI가 인덱스 검색 과정에서 GNI 보다 다소 더 많은 수의 단말 노드를 방문하기는 하지만 단말 노드의 클러스터링 정도가 MPI 경우 더 우수하기 때문인 것으로 판단된다. 반면 GNI는 여러 개의 인덱스를 검색해야 하므로 클러스터링 효과가 떨어진다. 그러나, 영역질의의 범위가 커질수록 클러스터링에 의한 MPI의 장점은 줄어드는데 왜냐하면 전반적으로 GNI보다 방문하는 단말 노드의 개수가 늘어나기 때문이다. 여기서 한



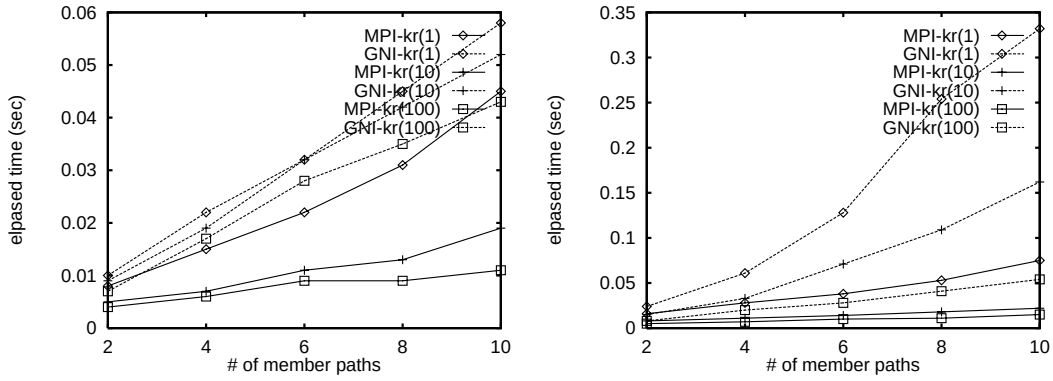
(a) kl = 8

그림 6: 일치 검색 성능 비교

가지 주목해야 할 점은 키값에 의해 정렬된 형태로 인덱스를 검색해야 하는 경우 MPI는 개개의 인덱스를 검색한 후 산출되는 결과를 병합하여야 하는 과정이 필요하다는 점이다. 이러한 상황에서는 MPI의 경우 그림에서 제시된 소요 시간보다 더 많은 시간을 필요로 하게 될 것이다. 본 실험은 모두 역방향 인덱스만을 대상으로 하였으나 순방향 인덱스의 경우에는 키의 중복도가 커지므로 다중 경로가 더욱 유리할 것으로 판단된다.

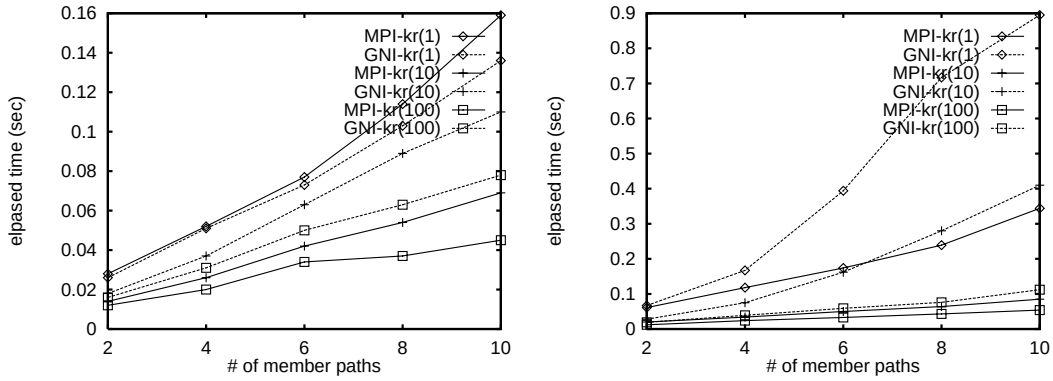
그림 8은 MPI와 GNI를 '순수 경로 인덱스'로 사용한 경우에 대해 성능을 비교한 것이다[3]. 이 경우 인덱스 경로는 'C_t.*.C_p'에 해당한다. 그리고 객체 식별자를 인덱스 키로 사용하므로 $kl = 8$로 설정하였는데 이것은 대다수의 OODB에서 사용하는 키의 크기이다. 또한 같은 이유로 $kr = 1$만을 사용했으며 $f = 10$으로 설정하였다. 4장에서 설명하였듯이 순수 경로 인덱스 형태에 대해서는 일치 검색만이 의미가 있기 때문에 일치 검색 연산 성능만을 비교한다. 다만 이전 실험에서는 1회의 일치 검색 연산에 대한 성능을 비교하였지만 이번 실험에서는 매번 임의의 인덱스 키값(또는 객체 식별자)을 검색하는 일치 검색 연산을 100 번, 1,000 번, 10,000 번 수행한 결과를 보인 것이다. 이것은 'C_t.*.C_p' 경로를 탐색하기에 앞서 C_p 클래스 인스턴스 전체에 대한 검색을 통해 그것들의 0.1%, 1%, 10%가 선택된 상황을 가정한 것이다. 그림 8에서 보듯이 선택율에 상관없이 어느 상황에서나 MPI가 GNI보다 우수한 성능을 보이는데 특히 N의 값이 비교적 큰 영역($6 \sim 10$)에서는 거의 10배 이상의 차이를 보인다. 이것은 MPI가 일치 검색에서 우수한 성능을 나타내는 것과 동일한 요인에 기인하며 더불어 다음과 같은 추가적인 요인 때문인 것으로 파악된다. 즉 MPI는 인덱스를 하나만 사용하기 때문에 인덱스 중간 노드들의 개수가 더 적은 반면 인덱스 검색 과정에서 동일한 중간 노드들을 방문할 가능성이 GNI의 경우보다 더 크다는 것이다. 이것은 곧 MPI의 경우 인덱스 중간 노드들이 버퍼에 캐쉬되어 있을 가능성을 높이고 그에 따라 검색 성능이 더욱 향상된 것으로 볼 수 있다.

[3] 세로축은 로그 스케일(log scale) 표시되어 있다.



(a) kl = 8, rqr = 1%

(b) kl = 64, rqr = 1%



(c) kl = 8, rqr = 5%

(d) kl = 64, rqr = 5%

그림 7: 영역 검색 성능 비교

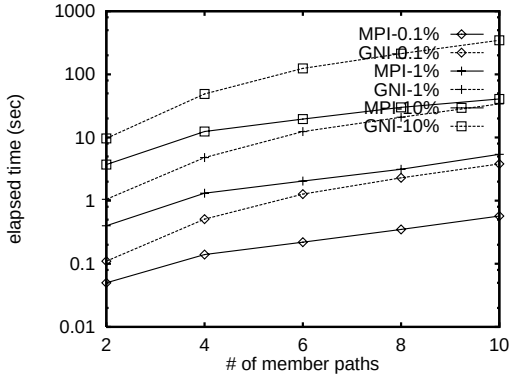


그림 8: 순수 경로 인덱스 성능 비교

6 결론

최근 OODB가 XML 데이터의 저장 장치로서 각광받고 있는데, XML 데이터에 대한 검색은 기존 OQL에서는 제공하지 않던 기능인 확장 경로식을 사용할 수 있다. 본 논문에서는 확장 경로식을 포함하는 질의의 성능을 향상시킬 수 있는 새로운 인덱스 기법으로 다중 경로 인덱스 방법을 제안하였고 그 성능을 평가하였다.

다중 경로 인덱스는 확장 경로식의 각 멤버 경로에 대해 경로 식별자를 할당하고 B^+-트리의 단말 노드 레코드를 변경시켜 다중 경로에 대한 인덱스가 가능하도록 하였다. 다중 경로 인덱스는 확장 경로식으로 표현되는 질의에 대한 검색 성능을 향상시킬 수 있다. 또한 널리 사용되고 있는 B^+-트리를 크게 수정하지 않고 사용할 수 있어 새로운 인덱스 기법에 따른 부가적인 동시성 제어 기법이나 고장 회복 기법이 필요 없으므로 실용성이 우수하다. 또한 다수의 경로에 대한 인덱스를 통합적으로 제공함으로써 인덱스 관리가 용이하고 확장 경로뿐만 아니라 개개의 경로에 대한 검색 기능을 제공할 수 있으며 경로 식별자를 이용하는 방식은 경로 인덱스, ASR과 같은 기존의 단일 경로 인덱스 기법에 쉽게 응용해서 사용할 수 있다.

추후에는 DTD 내에 사이클(cycle)을 포함하는 경로에 대한 인덱스 기능(데이터 수준의 사이클은 제외)과 정규식으로 주어진 경로식에 대한 인덱스 기능을 제공하기 위한 연구가 진행될 예정이다.

참고 문헌

[1] Neil Bradley. *The XML Companion, Second Edition.* Addison-Wesley, 1999.

[2] Alin Deutsch, Mary F. Fernandez, and Dan Suciu. Storing semistructured data with stored. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 431–442, 1999.

[3] Vassilis Christophides, Serge Abiteboul, Sophie Cluet, and Michel Scholl. From structured documents to novel query facilities. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 313–324, 1994.

[4] Poet Software Corp. *XML - The Foundation for the Future.* http://www.poet.com/products/cms/white_papers/xml/index.html.

[5] Zhaohui Xie and Jiawei Han. Join index hierarchies for supporting efficient navigations in object-oriented databases. In *Proceedings of the International Conference on Very Large Data Bases*, pages 522–533, 1994.

[6] Elisa Bertino and Won Kim. Indexing techniques for queries on nested objects. *IEEE Trans. on Knowledge and Database Eng.*, 1(2):196–214, 1989.

[7] Alfons Kemper and Guido Moerkotte. Access support in object bases. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 364–374, 1990.

[8] Jayavel Shanmugasundaram, He Gang, Kristin Tufte, Chun Zhang, David DeWitt, and Jeffrey Naughton. Relational databases for querying xml documents: Limitations and opportunities. In *Proceedings of the International Conference on Very Large Data Bases*, pages 302–314, 1999.

[9] Daniela Florescu and Donald Kossmann. Storing and querying xml data using an rdmbs. *IEEE Data Engineering Bulletin*, 22(3):27–34, 1999.

[10] Kien A. Hua. Object skeletons: An efficient navigation structure for object-oriented database systems. In *Proceedings of the International Conference on Data Engineering*, pages 508–517, 1994.

[11] Ehud Gudes. A uniform indexing scheme for object-oriented databases. In *Proceedings of the International Conference on Data Engineering*, pages 238–246, 1996.

[12] Tova Milo and Dan Suciu. Index strucutres for path expressions. In *Proceedings of the International Conference on Database Theory*, pages 277–295, 1999.

[13] Alfons Kemper and Guido Moerkotte. Access support relations: An indexing method for object bases. *Information Systems*, 17(2):117–145, 1992.

[14] Roy Goldman and Jennifer Widom. Dataguides: Enabling query formulation and optimization in semistructured databases. In *VLDB Journal*, pages 436–445, 1997.

[15] David Maier, Jacob Stein, Allen Otis, and Alan Purdy. Development of an object-oriented dbms. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications(OOPSLA)*, pages 472–482, 1986.