시그니처 기반 블럭 탐색을 통한 XML 질의 최적화 기법

(An XML Query Optimization Technique by Signature based Block Traversing)

박 상 원 † 박 동 주 †† 정 태 선 † 김 형 주 †††
(Sangwon Park) (Dong-Joo Park) (Tae-Sun Chung) (Hyoung-Joo Kim)

요 약 인터넷에서 사용되는 많은 데이타들이 XML로 표현되고 있는 추세이다. 이러한 XML 데이타는 트리 형태로 표현되므로 이것을 저장하고 질의하는 시스템으로는 그 모델링 능력 때문에 객체 저장소가 적합하다. 객체 저장소에서 XML의 각 노드는 객체로 저장된다. XML 질의의 특징은 정규 경로식으로 표현되는 것이며 이것은 XML 트리의 각 객체를 탐색하면서 처리된다. 정규 경로식을 지원하기 위하여 여러 인덱스들이 제안되었지만 이러한 인덱스들은 디스크 공간이라는 제약 때문에 모든 가능한 경로에 대한 인덱스를 제공하지는 못한다. 이러한 상태에서 정규 경로식을 잘 지원하기 위해서 블럭 탐색과 시그니처 방법을 이용하여 질의를 효과적으로 처리하는 최적 객체 탐색 기법을 제안하였다. 시그니처는 트리의 각 노드에 시그니처를 첨가하여 탐색 범위를 줄이는 것이다. 블럭 탐색은 한 페이지 내에 있는 접근 가능한 객체들을 미리 처리함으로써 디스크 I/O를 줄이는 것이다. 이와 같은 두가지 방법을 같이 이용하면 일반적인 질의 처리보다 월등히 나은 성능을 보인다는 것을 실험을 통하여 보였다.

키워드 : XML 질의 처리, 블록 탐색, 시그니처

Abstract Data on the Internet are usually represented and transfered as XML. The XML data is represented as a tree and therefore, object repositories are well-suited to store and query them due to their modeling power. XML queries are represented as regular path expressions and evaluated by traversing each object of the tree in object repositories. Several indexes are proposed to fast evaluate regular path expressions. However, in some cases they may not cover all possible paths because they require a great amount of disk space. In order to efficiently evaluate the queries in such cases, we propose an optimized traversing which combines the signature method and block traversing. The signature approach shrink the search space by using the signature information attached to each object, which hints the existence of a certain label in the sub-tree. The block traversing reduces disk I/O by early evaluating the reachable objects in a page. We conducted diverse experiments to show that the hybrid approach achieves a better performance than the other naive ones.

Key words : XML query processing, block traversing, signature

1. 서 론

XML은 인터넷에서 데이타를 표현하거나 전달하는 수단으로 널리 표준으로 채택되고 있다. 많은 데이타들이 XML로 표현되고 처리되어야 함에 따라 이를 효율적으로 처리하기 위하여 데이타베이스 시스템이 사용되어야 한다. XML은 기존의 데이타 모델과는 다른 그래프 형태의 반구조적(semistructured)[1, 2]인 특징을 가지고 있다. 기존 데이타베이스 모델과 다른 이러한 상이한 구조로 인하여 새로운 저장 방법 및 질의 처리 모델이 필요하며, 최근 몇년 동안 데이타베이스 분야에서 깊이 연구되어 왔다. XML 데이타는 트리 형태로 표현되며 관계형 데이타 모델과 비교하여 모델 표현력이 좋은

· 이 논문은 2001년도 두뇌한국21 사업에 의하여 지원되었음.
† 비 회 원 : 서울대학교 컴퓨터공학부
swpark@oopsla.snu.ac.kr
tschung@oopsla.snu.ac.kr
†† 비 회 원 : 삼성전자 정보통신총괄 통신연구소 연구원
djpark99@samsung.co.kr
††† 종신회원 : 서울대학교 컴퓨터공학부 교수
hjk@oopsla.snu.ac.kr
논문접수 : 2001년 5월 9일
심사완료 : 2001년 10월 24일

객체지향 모델로 잘 표현될 수 있다. 그러므로 Lore[3], eXcelon[4] 및 PDOM[5]과 같은 객체 저장소(object repository)는 XML 데이타를 저장하기 위한 적합한 시스템이다. XML 질의는 정규 경로식을 포함하고 있는 특징이 있으며, 이러한 질의는 객체 형태로 저장된 트리의 각 노드를 탐색하면서 처리된다. XML 질의를 효율적으로 처리하기 위하여 트리의 탐색 범위를 줄이고 페이지 I/O를 줄이는 것은 아주 중요하다.

```
SELECT x.(telephone|company.*.tel)
FROM person x;
```

위 질의는 Lorel[6]과 유사한 XML 질의의 한 예로서 사람들의 전화번호를 구하는 것이다. 이 질의에는 XML-QL[7]이나 XQL과 같은 일반적인 XML 질의에서 지원되는 정규 경로식(regular path expression)[6, 7, 8]이 있다. 예제에서 company.*로 인하여 person의 하위 노드 대부분을 방문하여야 하는 것과 같이 *와 같은 문장은 질의를 처리하기 위한 탐색 범위를 매우 크게 하는 문제가 있다. 그러므로 이러한 문제를 해결하기 위하여 정규 경로 인덱스(regular path index)들이 연구되어 왔다.

객체지향 데이타베이스에서 경로식(path expression)을 처리하기 위하여 경로 인덱스[10]가 제안되었다. 하지만 디스크 공간의 제약 때문에 가능한 모든 경로에 대하여 인덱스를 생성할 수 없다. 정규 경로식을 처리하기 위하여 1-index[11]가 제안되었다. 이것은 *P.x*와 같은 경로식을 지원하는데 *P*는 루트에서 시작하는 정규 경로식이다. 또한 임의의 위치에서 시작하는 정규 경로식을 위하여 2-index[11]가 제안되었는데 이것은 **.x.P.y*와 같이 정규 경로식 *P*가 임의의 위치에서 시작할 수 있다. 하지만 최악의 경우 2-index에서의 노드의 갯수는 데이타 노드의 제곱배가 될 수 있기 때문에 T-index[11]로 포괄할 수 있는 범위를 제한하였다. T-ndex는 **.person.x.P.y*와 같이 시작 노드를 **.person*의 하위 노드로 제한한 것이다.

이와 같이 실제적인 이유 때문에 어떤 데이타들은 인덱스를 통하여 구할 수가 없고 직접 데이타 노드를 탐색하여야 한다. 그러므로 정규 경로식을 빠르게 수행하기 위하여 새로운 방법이 제시되어야 한다. 우리는 이와 같이 인덱스를 사용할 수 없는 경우에 탐색 범위를 줄이기 위하여 시그니처 방법[12, 13]을 개발하였다. 하지만 탐색 범위를 줄이더라도 객체 저장소에서 객체들이 클러스터링이 되어 있지 않으면 페이지 I/O가 많이 일어난다. 이와 같은 페이지 I/O를 줄이기 위하여 같은 페이지에 저장된 객체를 다른 객체들 보다 먼저 처리함으로서 페이지 I/O를 줄이는 *블럭 탐색*(block traversing)을 제안하였다. 시그니처로 탐색 범위를 줄이고 블럭 탐색으로 불필요한 페이지 I/O를 줄임으로서 객체 저장소에 저장된 객체의 분포와 관계없이 굉장히 많은 페이지 I/O를 줄일 수 있었다. 이와 같은 방법은 반구조적 인덱스의 페이지 I/O를 줄이는데도 사용될 수 있다.

본 논문의 구성은 다음과 같다. 2절에서 관련 연구에 관하여 설명하고 3절에서 데이타 모델과 질의어, 질의 처리 방법 및 OID 테이블에 대하여 설명한다. 4절에서 본 논문에서 제안한 DOM 트리를 탐색할 때 페이지 I/O를 줄이는 세가지 방법에 대하여 설명한다. 5절에서 실험 결과를 설명하고 6절에서 결론을 맺는다.

2. 관련 연구

반구조적 데이타[1, 2]는 그래프로 표현된다. 반구조적 데이타에 대한 질의어는 OQL[14]이나 XSQL[15]과 같은 객체 지향 데이타베이스 질의어로부터 영향을 받았다. OQL이나 XSQL은 경로식으로 질의의 표현력을 증진시켰다. 하지만 반구조적 데이타는 스키마 정보가 없기 때문에 이러한 질의도 적합하지 않다. 스키마 정보가 제공되더라도 그 자신의 데이타에 의해 구조가 변할 수 있다.

이러한 문제를 해결하기 위하여 정규 경로식이 반구조적 데이타에 대한 질의[6, 8, 9]로 사용되었다. 이러한 정규 경로식을 빠르게 처리하기 위하여 반구조적 데이타에 대한 인덱스[11, 16, 17]가 제안되었다. 이것은 XML 데이타의 구조와 오토마타를 결합한 구조이다. 즉 정규 경로식에 대하여 해당 오토마타 그래프를 탐색하면서 목적 객체를 찾아낸다. 이러한 인덱스는 저장 공간을 많이 차지하기 때문에 가능한 모든 경로에 대하여 인덱스를 생성하기가 곤란하다.

기존 데이타베이스로 XML 문서를 효율적으로 저장하기 위한 방법이 많이 연구되어 왔다[18, 19, 20, 21]. 이러한 방법들은 주로 XML 문서를 관계형 데이타베이스에 저장하기 위한 스키마 생성과 질의 최적화를 위한 질의 변환에 촛점을 두었다. [22]에서는 DTD 정보가 없는 XML 문서를 위하여 데이타로부터 DTD 정보를 효율적으로 추출하는 방법을 제시하였다.

XML 문서를 저장하기 위한 객체 저장소로 Lore[3], eXcelon[4] 및 PDOM[5] 등이 있다. 객체 지향 데이타 모델의 뛰어난 모델링 능력 때문에 각 노드를 객체로 저장함으로서 XML 문서의 원 구조를 변경시키지 않고 저장할 수 있다.

[23]은 객체지향 데이타베이스을 위한 시그니처 방법

을 제시하였다. 이것은 OQL의 경로식을 이용하여 질의를 처리할 때 특정 경로에 속하는 특정 애트리뷰트들에 대하여 해쉬값을 구한 후 이를 비트 연산으로 OR 하여 시그니처를 구한 후 이를 OID 테이블에 저장한다. 이렇게 하면 a.c.c = 'kim' 과 같은 질의를 처리할 때 OID 테이블에서 객체 a의 시그니처가 kim의 해쉬값과 같은지를 비교한다. 이렇게 하면 경로식을 따라 모든 객체를 디스크에서 읽지 않아도 되기 때문에 수행 속도가 빨라진다. 하지만 이러한 방식은 XML 질의의 특징인 정규 경로식을 지원하지 못한다. [23]는 경로식의 특정 애트리뷰트 값을 비교하는 것이지만 XML 질의는 정규 경로식을 따라가면서 질의를 처리해야 하는데 정규 경로식에 해당하는 객체들의 수를 줄이는 것이 중요한 점이다.

관계형 데이타베이스에서 여러개의 conjunctive 질의를 위한 시그니처 방법도 있다[24]. 이것은 SQL의 WHERE 절에서 특정 애트리뷰트의 값이 어떤 값인지를 미리 B 트리에 저장해 두고 질의를 처리할 때 이를 이용하면 질의를 처리하기 위하여 읽어야 하는 디스크 I/O를 줄일 수 있다는 원리이다.

위의 두가지 방법은 모두 특정 애트리뷰트의 값에 대하여 시그니처를 구하였다. 본 논문에서 처리하고자 하는 정규 경로식에 대한 시그니처 방법은 특정 애트리뷰트가 아니라 정규 경로식에 참여하는 각 경로에 대하여 시그니처를 만드는 방법이다. 그러므로 이전의 방법은 정규 경로식을 처리하는데 시그니처를 사용할 수 없지만 정규 경로식에 시그니처를 만들면 정규 경로식을 처리할 때 시그니처를 이용하여 탐색 범위를 줄일 수 있다.

3. 준비 단계

이번 절에서는 XML 데이타의 데이타 모델과 질의어 및 질의 처리 방법에 대하여 설명한다. 3.2절에서 질의를 NFA로 변환하는 방법과 NFA를 이용하여 정규 경로식을 처리하는 방법에 대하여 설명한다. 3.3절에서 본 논문에서 제안한 방법을 구현하기 위하여 객체 저장소의 OID 테이블에 대하여 설명한다.

3.1 데이타 모델과 질의어

XML 데이타 모델은 순서가 있는 형제 노드를 가진 트리 형태로 표현된다. DOM은 XML 데이타에 대한 표준 인터페이스로서 이 API를 통하여 트리를 탐색한다. 본 논문에서 사용한 데이타 모델은 DOM이다.

그림 1은 DOM 트리의 예로서 각 노드는 그 자신의 부모 노드, 자식 노드 혹은 형제 노드를 탐색할 수 있다. 각 노드는 객체 저장소에 객체로 저장된다. 각 객체는 OID를 가지며 그림 1에 '&'로 표기되었다. DOM에는 엘리먼트 노드와 애트리뷰트 노드 및 텍스트 노드가 있으며 각각은 이름이나 값을 가지고 있으며 이러한 객체들은 디스크 페이지에 저장되어 있다. 예를 들어 그림 1에서 객체 &1, &2, &6은 페이지 A에 저장되어 있다. 다음 정의들은 본 논문에서 설명하는데 유용한 것들이다.

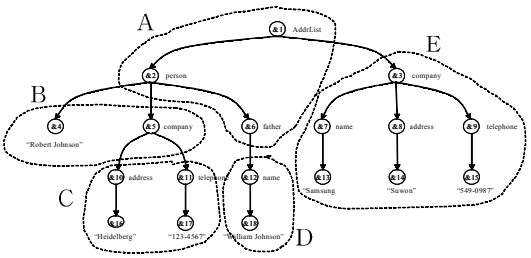


그림 1 DOM 그래프

정의 3.1 (레이블 경로) DOM 객체 o의 레이블 경로는 점으로 분리된 순서가 있는 하나 혹은 그 이상의 레이블 $l_1.l_2..l_n$이다. 즉 노드 o에서 n까지 경로상에 있는 객체 $(n_1...n_n)$를 따라갈 수 있다. 이때 노드 n_i의 레이블은 l_i이며 노드의 타입은 엘리먼트 혹은 애트리뷰트이다.

정의 3.2 (정규 경로식) 정규 경로식은 레이블 경로에 정규 경로식이 있는 것을 말한다.

본 논문에서 질의는 *, + 및 ?와 같은 와일들 카드(wild card)를 지원하는 정규 경로식이다. 스캔 연산자는 주어진 정규 경로식을 만족하는 객체를 찾아 반환하는 것으로서 이와 같은 XML 질의는 트리의 각 노드를 탐색함으로써 처리된다.

3.2 NFA를 이용한 정규 경로식 처리

본 논문에서 정규 경로식을 처리하기 위하여 이것을 NFA(non-deterministic automata)로 변환한다. 일반적으로 모든 정규식은 NFA로 변환할 수 있다[25]. 정규 경로식도 정규식이므로 NFA로 변환이 가능하다. 그러므로 아무리 복잡한 정규식도 그림 3에 나타난 것과 같

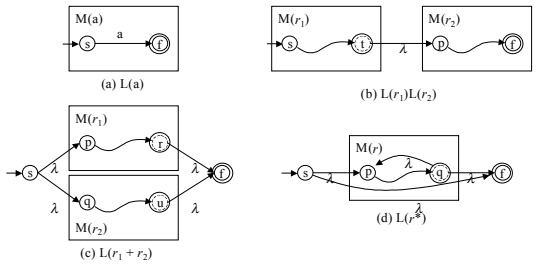


그림 2 NFA

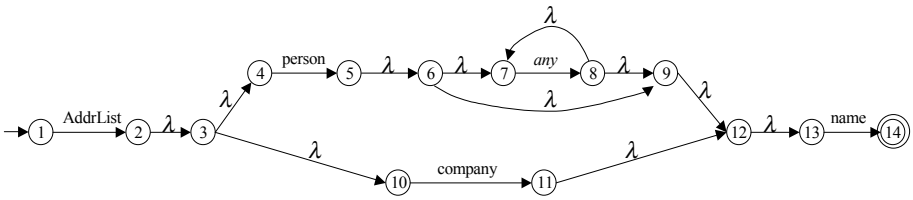


그림 3 addr.(person.*|company).name의 NFA

이 L(r_1)L(r_2), L(r_1 + r_2) 및 L(r*)의 조합으로 나타낼 수 있다[25]. L(r?)와 L(r+)은 L(r*)의 변형으로 표현할 수 있다. L(r?)은 그림 2(d)에서 상태 노드 q에서 상태 노드 p로 가는 λ 에지를 제거하여 얻어진다. L(r+)은 L(r*)에서 상태 노드 s에서 f로 가는 에지를 제거하면 된다. 본 논문에서 사용하는 정규 경로식은 addr.(person.*|company).name이고 이 질의에 대한 NFA는 그림 3에 나타나 있다.

정의 3.3 (상태집합(state set)) 상태 집합은 NFA의 상태 노드들의 집합을 말한다. 여기서 각 원소는 특정 레이블 경로에 의해 NFA에서 전이된 경로들의 결과를 뜻한다.

NFA에서의 상태 천이는 DOM 객체의 레이블에 의해 결정되고 그 결과로 상태 집합을 구할 수 있다. 상태 천이를 통해 최종 상태(final state)에 도달하면 그 객체의 상태 집합에 최종 상태가 있기 때문에 그 객체가 질의 결과 집합의 한 원소가 된다. 만약 상태 집합이 공집합이면 그 객체의 서브 트리는 방문하지 않아도 된다. 그 이유는 천이를 할 상태 노드가 상태 집합에 없기 때문이다.

예제 3.1 정규 경로식 addr.(person.*|company).name의 그림 1에 대한 결과 집합은 R = {&7, &12} 이다. 처음 상태 집합 SS는 그림 3의 시작 상태 노드인 {1}이다. DOM 트리에서 루트 객체 &1을 가져오면 레이블 경로는 addr이 되고 SS는 {4, 10}이 된다. 그런 다음 객체 &2를 가져오면 레이블 경로는 addr.person이 되고 SS는 {7, 13}이 된다. 객체 &12를 가져오면 SS는 {7, 14}가 되고 상태 집합 SS에 최종 상태 노드 14가 있게 된다. 그러므로 객체 &12는 결과 집합의 객체가 되므로 R = {&12} 이다. 이와 같은 연산을 계속하면 주어진 정규 경로식의 결과 R이 얻어진다. 이 연산은 상태 집합 SS가 공집합이 되면 멈춘다. 트리는 깊이 우선 방식으로 탐색한다. 그러므로 객체 &6이 처리되기 전에 &4, &5와 그 자식 객체들이 먼저 처리된다. 페이지 A는 객체 &4, &5를 처리한 후 버퍼에서 나갈 수 있다. 이와 같은 문제 뿐만 아니라, person.* 때문에 person의 자식 객체들 대부분이 방문되어야 한다.

3.3 객체 저장소의 OID 테이블

객체 저장소의 각 객체는 그 자신의 물리적인 위치 정보를 가지고 있는 OID가 있다. OID에는 두가지 종류가 있는데, 첫째는 논리적인 OID(LOID)이고 다른 하나는 물리적인 OID(POID)이다[26]. 논리적인 OID는 시스템에 의해 만들어지는 번호(pseudo number)이다. 이러한 경우에 LOID와 그 객체의 물리적인 위치를 매핑하는 OID 테이블이 있어야 한다. 본 논문에서는 그림 4에 나타난 것과 같이 객체 저장소는 LOID를 가지고 OID 테이블이 있다고 가정한다. POID는 객체의 물리적인 위치를 뜻하므로 OID 테이블이 불필요하다. 본 논문에서는 두가지 테크닉을 제안한다. 하나는 블럭 탐색이고, 다른 하나는 시그니처 기반 탐색이다. LOID나 POID 모두 블럭 탐색에 사용할 수 있다.

그림 4에서 보는 바와 같이 OID 테이블에 각 객체의 시그니처를 덧붙였다[23]. 이것은 어떤 레이블이 서브 트리에 존재하는 지 알아내는 힌트를 제공한다. 만약 POID를 사용할 경우에는 OID 테이블이 없으므로 시그니처는 각 객체에 저장되어야 한다[12, 13].

객체 n의 해쉬 값을 H_n이라 하고 그 객체의 시그니처를 S_n이라 하면 $S_n = \bigvee_{child\ i\ of\ n} H_i$ 이다. 이것은 객체 n의 모든 자식 노드의 해쉬 값을 OR 연산한 것이다. 객체 i의 서브 트리에 레이블 l을 가진 객체가 존재하는 지는 $H_l \wedge S_i$을 연산하여 추측할 수 있다. 만약 $H_l \wedge S_i \equiv H_l$이면 서브 트리에 레이블 l을 가진 객체가 존재

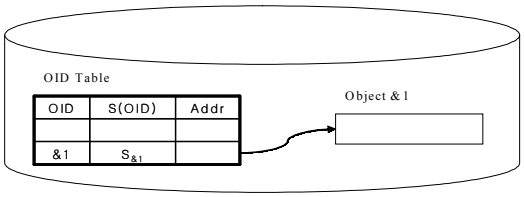


그림 4 OID 테이블

할 가능성이 있다. 만약 $H_l \wedge S_i \neq S_i$이면 서브 트리에 그러한 객체가 없다는 것을 확신할 수 있다.

4. 새로운 객체 탐색 방법

이번 장에서는 블럭 탐색과 시그니처 기반 탐색에 대하여 설명한다. 또한 이 두가지

방법을 결합한 최적화 탐색을 소개한다.

4.1 블럭 탐색

DOM 트리에서 각 노드는 객체 저장소에 객체로서 저장된다. 객체가 처음 저장될 때는 클러스터링이 되어 저장된다. 하지만 갱신 연산이 많이 일어나게 되면 객체들은 여러 페이지에 걸쳐서 흩어지게 된다. 결과적으로 근처에 있는 객체들이 여러 페이지에 저장되게 된다. 이러한 객체를 읽어들이면 많은 페이지 폴트를 일으킨다. 예를 들어 그림 1의 객체 &2, &4, &5, &6은 순서대로 접근된다. 객체 &2 및 &6은 페이지 A에 저장되어 있고 객체 &4와 &5는 페이지 B에 저장된다. 페이지 B에 있는 객체들을 처리한 후 객체 &6과 그 자식 객체들을 깊이 우선 탐색으로 방문하면서 처리하여야 한다. 객체 &2의 자식 객체들을 방문한 후 페이지 A는 버퍼 관리자에 의해 버퍼에서 제거될 가능성이 있다. 이것은 페이지 폴트를 유발한다. 이것은 객체들간의 지역적인 정보(locality information)을 무시하기 때문에 발생한다.

관계형 데이타베이스의 중첩 블럭 조인[27]과 같이 읽혀진 페이지 내의 모든 객체를 한번에 모두 처리할 수 있다면 가장 좋은 방법이다. 하지만 정규 경로식을 처리하기 위해서는 부모 객체로부터의 상태 집합을 알아야 한다. 즉, 모든 객체는 부모 객체가 처리되지 않으면 처리될 수 없음을 뜻한다. 객체 저장소에 저장된 객체는 OID를 가지고 있으며 이것은 디스크의 물리적인 위치 정보를 뜻한다. 그러므로 이러한 OID 정보를 이용하여 두 객체가 같은 페이지에 저장되어 있는지 알아낼 수 있다. 또한 DOM 트리 내의 객체는 그들의 자식 객체들에 대한 OID를 가지고 있다. 이것은 어떤 자식 객체가 부모 객체와 같은 페이지에 저장되어 있는지 알아낼 수 있다는 뜻이다. 그러므로 우리는 이러한 객체들을 미리 처리함으로써 많은 페이지 폴트를 줄일 수 있다.

정의 4.1 (이웃 객체(next-door object)) 두 객체 o_i와 o_j가 같은 페이지에 저장되어 있으면 o_i와 o_j는 서로 이웃 객체이다.

알고리즘 1은 정규 경로식을 만족하는 객체를 반환하는 스캔 연산자이다. 이 함수를 수행하기 전에 (루트 객체의 OID, { }) 쌍이 큐에 저장되어야 한다. 여기서 큐

알고리즘 1 block::next()

```
1:  /* SS is the state set of NFA */
2:  node ← Queue.remove()
3:  while node is not NULL do
4:    SS ← traverse NFA by the label of node
5:    /* if SS is empty then the sub-tree of node can not be
6:  query result */
7:    if SS is not empty then
8:      Queue.addChildren(node, SS)
9:      if a final state node ∈SS then
10:       return node
11:     end if
12:   end if
13:   node ← Queue.remove()
14: end while
    return NULL
```

는 (OID, 부모 객체의 상태 집합)을 쌍으로 저장하는 것이다. 이웃 객체를 빠르게 찾아서 읽어 들이기 위하여 해쉬 테이블을 구현되었다. 경로 스택(path stack)은 루트부터 현재 처리하고 있는 객체까지의 경로를 관리하기 위한 것이다. 경로 스택의 최상위 객체 o_t는 방금 처리된 객체를 뜻한다. 함수 remove는 객체 o_t의 이웃 개체 중 하나를 반환한다. 만약 o_t의 이웃 객체가 큐에 없다면 같은 페이지에 속한 객체가 가장 많은 페이지에 속한 임의의 객체 하나를 반환한다. 즉 페이지 번호에 의해 큐에 저장된 객체를 그룹지을 수 있다. 객체 &i의 상태 집합을 SS_i라 하고 &i의 자식 객체 중 하나를 c_i라 가정하자. 함수 addChildren은 이웃 객체를 읽어들이기 위하여 큐에 (c_i의 OID, SS_i) 쌍을 저장한다.

예제 4.1 객체 &i의 상태 집합을 SS_i라 하자. 그림 1의 루트 객체가 읽혀지고 그림 3의 NFA를 탐색할 때 상태 집합 SS_1는 {4, 10}이 된다. 이것의 자식 객체와 상태 집합의 쌍인 (&2, SS_1)과 (&3, SS_1)이 큐에 삽입된다. &1의 이웃 객체는 &2이기 때문에 함수 remove는 &2를 리턴한다. &2를 처리한 후 SS_2는 {7, 13}이 된다. 그러므로 (&4, SS_2), (&5, SS_2), (&6, SS_2)가 큐에 삽입된다. 함수 remove는 객체 &6을 반환하므로 비록 &2의 첫 번째 자식 객체가 &4일지라도 &4에 앞서 &6이 먼저 처리된다. 객체 &6이 처리된 후 큐에 (&12, SS_6)이 삽입된다. 그러면 큐에는 객체 &6이 없게 된다. 이때 본 논문에서는 큐에 저장되어 있는 객체들의 페이지 중에 큐에서 가장 많은 객체를 가진 페이지 중 임의의 한 객체를 반환하도록 하였다. 이러한 방법에 의하여 remove 함수는 객체 &2를 반환한다.

4.2 시그니처 탐색

우리는 XML 질의를 빠르게 처리하기 위하여

s-NFA[12, 13]이라 불리는 시그니처 기반 탐색 방법을 개발하였다. 이러한 시그니처 기반 탐색은 탐색 공간을 줄여서 질의 처리 시 페이지 I/O를 줄인다. 이러한 기법을 객체 저장소에 적용하기 위하여 그림 4에 나타난 OID 테이블에 시그니처 컬럼을 추가하는 방법을 이용하였다.

정규 경로식은 *, +, ?와 같은 와일드 카드(wild card)를 사용할 수 있다. 스캔 연산자는 질의를 처리할 때 주어진 정규 경로식을 만족하는 객체를 찾기 위한 것이다. person.*.name과 같은 질의의 경우 * 때문에 person의 모든 자식 노드는 방문하여야 한다. 만약 미리 질의를 만족하는 객체가 서브 트리 내에 존재하는지를 판단할 수 있다면 그래프를 방문하지 않을 수 있으며 트리의 탐색 공간을 줄일 수 있다.

정의 4.2 (NFA 경로) 비결정적 오토마타(NFA)의 한 상태 노드에서 최종 상태로 가는 경로를 NFA 경로라 한다. 즉 한 상태 노드의 NFA 경로는 여러 개일 수 있다.

정의 4.3 (경로 시그니처(path signature)) NFA의 한 상태 노드 n의 경로 시그니처 PS_n은 다음과 같이 정의된다. PS_n = {x | x는 NFA의 상태 노드 n의 한 NFA 경로에 나타나는 모든 레이블의 시그니처 값을 비트 연산 OR 한 값}.

경로 시그니처는 NFA 경로에 있는 모든 레이블에 대하여 해쉬 값을 OR 한 비트 값이다. NFA의 한 상태 노드 n에서 최종 상태 노드로 가는 경로는 여러 개가 있기 때문에 상태 노드 n의 NFA 경로는 여러 개가 있다. 그러므로 상태 노드 n의 경로 시그니처 PS_n은 집합이다. s-NFA는 각 상태 노드가 경로 시그니처를 가진 NFA를 말한다.

만약 상태 노드 n의 경로 시그니처를 PS_n이라 하고 객체 &i의 시그니처를 S_i라 하자. 또한 PS_n의 한 시그니처를 S_j라 하자. 만약 $S_j \wedge S_i \equiv S_j$라면 객체 &$i$의 서브 트리를 탐색할 때 최종 상태에 도달할 수 있다고 추측할 수 있다. 만약 그렇지 않다면 객체 &i의 서브 트리의 모든 객체를 방문하더라도 NFA의 최종 상태에 도달할 수 없다는 것을 확신할 수 있다. 그러므로 질의를 처리할 때 시그니처를 비교함으로써 트리 탐색 범위를 줄일 수 있는 것이다.

그림 2는 NFA의 여러 타입을 만드는 방법에 대하여 설명하고 있다. 그러므로 그림 2의 NFA의 경로 시그니처를 만들 수 있다면 아무리 복잡한 NFA에서도 경로 시그니처를 만들 수 있다. 경로 시그니처를 만드는 방법에 대해서는 아래에 설명되어 있다.

규칙 4.1 (L(a)) 그림 2(a)와 같이 원자값을 가지는 NFA는 시작 노드 s, 최종 노드 f 두개의 상태 노드를 가진다. 레이블 a의 해쉬값을 H_a라 하면 s 노드와 f 노드의 경로 시그니처 PS_s와 PS_f는 각각 PS_s = { H_a }, PS_f = { 0 }이다.

정의 4.2 (L(r_1 + r_2)) 그림 2(c)와 같이 두 개의 NFA가 | 로 연결된 정규식인 경우 PS_s, PS_f는 각각 $PS_s = PS_p \cup PS_q$, PS_f = { 0 } 이다.

정의 4.3 (L(r*)) 그림 2(d)와 같이 * 연산자인 경우의 경로 시그니처는 다음과 같다. 이 경우 *로 인하여 내부 시그니처는 PS_s에서 무시된다. 왜냐하면 내부를 거치지 않고 f 노드로 바로 진행할 수 있기 때문이다. 그러므로 PS_s = { 0 }, PS_f = { 0 }이다.

정의 4.4 (L(r+)) L(r+)는 그림 2(d)에서 s 노드에서 f 노드로 가는 에지를 삭제하면 된다. 이때 PS_s 노드를 제외하고는 규칙 4.3과 동일하다. $PS_s = PS_p$, PS_f = { 0 }.

L(r?)는 규칙 4.3과 동일하다.

정의 4.5 (L(r_1)L(r_2)) 두 개의 NFA가 합해지는(concatenation)되는 것으로서 M(r_1)의 모든 상태 노드의 경로 시그니처는 PS_p의 경로 시그니처를 가져야 한다. 이것을 시그니처 전파(signature propagation)라고 한다. 즉 시작 상태 노드에서 최종 상태 노드로 가려면 M(r_2)의 노드 p를 거쳐야 하므로 PS_p를 M(r_1)의 경로 시그니처에 반영해야 한다. 이것은 M(r_1)의 모든 상태 노드의 경로 시그니처와 경로 시그니처 PS_p를 카티션 프로덕트해야 한다는 것을 의미한다. 그러므로 M(r_1)의 각 노드 i의 경로 시그니처 PS_i은 다음과 같다. PS_i = { $(x \vee y)$ | PS'_i는 M(r_1)의 한 상태 노드의 경로 시그니처이다. 여기서 x는 PS'_i은 시그니처이고 y는 PS_p의 시그니처이다.}

알고리즘 2는 주어진 정규 경로식을 만족하는 객체를 반환하는 스캔 연산자이다. 이 스캔 연산자는 알고리즘

알고리즘 2 signature::next()

```
1: /* SS is the state set of s-NFA */
2: node ← get next node by DFS
3: while node is not NULL do
4:   SS ← forward(node, node.SS) /* using Signature */
5:   /* if SS is empty then the sub-tree of node can not be
     query result */
6:   if SS is empty then
7:     node ← get next node by DFS
8:   end if
9:   if a final state ∈ SS then
10:    return node
11:  end if
12: end while
13: return NULL
```

3에 기술되어 있는 forward를 호출한다. 이 함수는 s-NFA의 생태를 전진할 수 있는지 없는지를 OID 테이블의 시그니처와 경로 시그니처를 비교하여 결정한다.

알고리즘 3 forward(*node*, *SS*)

```
1: SS ← forward by label
2: for each state node n which can go forward by λ in SS do
3:   for each signature Si of PSn do
4:     if Si ∧ node.signature ≡ Si then
5:       m ← the state node moved from n by λ
6:       add m to SS
7:       break
8:     end if
9:   end for
10:  remove n from SS
11: end for
```

예제 4.2 예제 3.1의 질의를 s-NFA로 변환하면 위에서 정의한 규칙에 의하여 각 노드는 경로 시그니처를 가진다. 객체 &5의 시그니처 S_5는 $H_{"address"} \vee H_{"telephone"}$이다. 루트 객체부터 &5까지 탐색할 때 상태 집합 SS_5는 {7, 13}이 되고 &5의 경로 시그니처 PS_5는 $\{H_{"name"}\}$이 된다. 이때 S_5와 $H_{"name"}$ 사이의 비트 연산 AND의 결과가 PS_5의 경로 시그니처 중 하나인 $H_{"name"}$이 아니다. 이것은 &5의 서브 트리를 탐색해도 최종 상태로 갈 수 없음을 뜻한다. 그러므로 &5의 서브 트리는 질의 처리에서 제외된다.

4.3 최적화 탐색(the optimized traversing)

시그니처 기반 탐색은 시그니처 정보를 이용하여 서브 트리를 질의 처리에서 제외함으로써 탐색 공간을 줄인다. 블럭 탐색은 객체의 탐색 순서를 변경함으로써 페이지 I/O를 줄인다. 두가지 방법은 모두 페이지 I/O를 줄이는 것이지만 서로 다른 접근 방법을 사용하고 있다. 본 논문에서는 서로 상이한 두가지 방법을 결합하였다. 이것을 최적화 탐색이라 부른다.

알고리즘 4 optimal:next()

```
1: /* SS is the state set of NFA */
2: node ← Queue.remove()
3: while node is not NULL do
4:   SS ← forward(node, node.SS) /* using Signature */
5:   /* if SS is empty then the sub-tree of node can not
     be query result */
6:   if SS is not empty then
7:     Queue.addChildren(node, SS)
8:     if a final state node ∈ SS then
9:       return node
10:    end if
11:  end if
12:  node ← Queue.remove()
13: end while
14: return NULL
```

알고리즘 4는 최적화 탐색을 통하여 정규 경로식을 만족하는 객체를 반환하는 스캔 연산자이다. 이 알고리즘은 시그니처 정보를 이용하여 서브 트리를 제외하는 forward 함수를 제외하고는 알고리즘 1과 동일하다. 블럭 탐색을 이용하여 객체를 탐색하는데 그렇다 하더라도 블럭 탐색 도중 처리되는 객체의 하위 객체의 탐색 여부는 시그니처 정보를 이용하여 결정된다. forward 함수는 시그니처를 이용하여 하위 객체의 탐색 여부를 결정하는데 이 함수의 결과로 반환되는 상태 집합이 공집합이면 하위 객체를 탐색하여도 더 이상 질의 결과를 만족하는 객체가 없다는 의미가 된다. 그러므로 이 객체의 하위 객체는 더 이상 탐색하지 않는다.

최적화 기법은 블럭 탐색을 이용하여 객체를 탐색하는데 읽어들이는 페이지의 수를 줄이고 시그니처를 이용하여 불필요한 노드의 탐색을 줄인다. 결국 두 가지 방법은 전체적으로 페이지 I/O 횟수를 줄이는 역할을 한다.

5. 실험 결과

본 논문의 실험은 자바로 구현되었으며 질의는 메모리에서 처리된다. DOM의 각 노드는 객체로서 저장되며 이것들은 정규 경로식을 입력으로 스캔 연산자를 이용하여 가져온다. 스캔 연산자는 객체를 객체 캐쉬에 요구하며, 객체 캐쉬는 버퍼 관리자에게 객체가 속한 페이지를 요구한다. 우리는 성능 평가의 척도로 버퍼 관리자의 페이지 I/O 횟수만 측정하였다. 시그니처와 블럭 탐색을 위한 연산은 메모리에서 일어나는 것이므로 성능 평가에서 무시하였다. 페이지의 크기는 4K 바이트이고 버퍼의 크기는 20개이다.

객체 저장소에 처음 객체를 저장할 때는 클러스터링 되어 있는 상태로 저장된다. 하지만 시간이 지날수록 객체들은 많은 갱신 연산에 의하여 흩어지게 된다. 우리는 처음에는 깊이 우선 방식으로 객체를 저장하였다. 이것은 깊이 우선으로 완전 클러스터링이 된 상태로 저장한 것을 의미한다. 성능은 깊이 우선으로 저장하거나 넓이 우선으로 저장할 때 차이가 난다[12, 13]. 이러한 성능의 차이는 비 클러스터링 비율이 증가할 수록 작아진다. 구현의 관점에서 본다면 깊이 우선 방식이 큰 XML 화

표 1 XML 파일의 특성

	No. of Nodes	File Size
Shakespeare	537,621	7.5 Mbytes
Bibliography	19,854	247 Kbytes
The Book of Mormon	142,751	6.7 Mbytes

일을 저장할 때 더욱 적합하다. 본 논문에서 사용한 데이타는 세익스피어, 모르몬 경과 Michael Lay의 참고문헌을 XML로 변환한 것을 이용하였다. 이들 데이타에 대한 통계 정보는 표 1에 잘 나타나 있다.

표 2 실험에 사용된 질의

Q1	Shakespeare	PLAY.*[2].PERSONA
Q2	Shakespeare	*.TITLE
Q3	Bibliography	bibliography.paper.*[1].pages
Q4	Bibliography	*.author
Q5	The Book of Mormon	tstmt.*[1].(title|ptitle)
Q6	The Book of Mormon	*.chapter

실험을 위하여 표 2에 나타난 6개의 질의를 사용하였다. 이 질의에서 '*[2]'는 PLAY와 PERSONA 사이에 두개의 임의의 레이블을 가진 객체가 있다는 것을 의미한다. Q1, Q2, Q5는 특정 스트링을 만족하는 데이타를 찾는 질의이고, 나머지는 트리의 임의의 깊이에 존재하는 객체를 찾는 질의이다.

실험에 쓰인 스캔 연산자는 naive, signature based, block, optimized 네가지가 있다. naive 방법은 깊이 우선 탐색[1]으로 객체를 탐색하는 방법이다. signature based 방법은 시그니처를 이용하여 트리 내에 정규 경로식을 만족하는 레이블이 존재하는지 예측하면서 탐색하는 시그니처 기반 탐색 방법이다. block 방법은 트리를 탐색할 때 이웃 객체를 우선적으로 탐색하는 블럭 탐색 방법이다. 이것은 객체 탐색 순서를 변경하여 페이지 I/O를 줄이는 것이다.

optimized 방법은 이 두가지 방법을 결합한 최적화 탐색 방법이다.

그림 5는 성능 평가 결과를 보여준다. 결과로 Q1, Q3, Q5 세가지만 보여주는데 이것은 Q2, Q4, Q6의 결과가 Q3과 유사하기 때문에 생략하였기 때문이다. 그림 5에서 비 클러스터링 비율이 0일 때는 객체가 깊이 우선 방식으로 완전 클러스터링 되어 있는 것을 의미하며 x 축의 비율에 따라 객체는 흩어지도록 하였다. 큐의 크기와 시그니처의 크기는 각각 4,000개 및 4 바이트로 고정하였다. 각각의 비 클러스터링 비율에 대하여 페이지 I/O를 측정하였다.

그림 5에서 보는 바와 같이 블럭 탐색 방법을 이용하

1) 넓이 우선 탐색으로 탐색할 수도 있다. 하지만 실제적인 면에서 질의를 처리할 때 많은 자식 객체를 큐에 삽입하여 처리한다는 것은 거의 불가능하다.

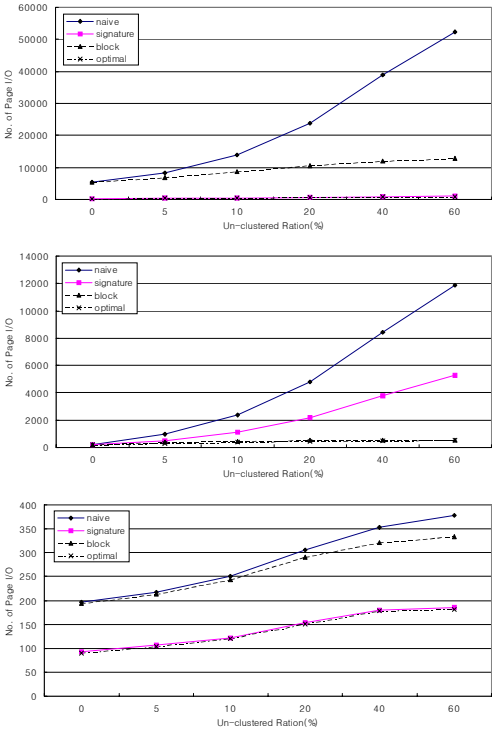
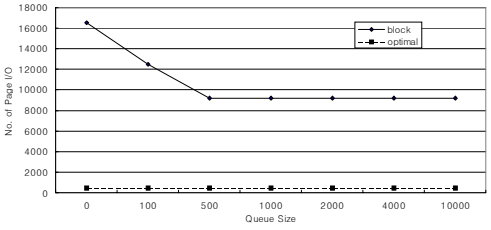
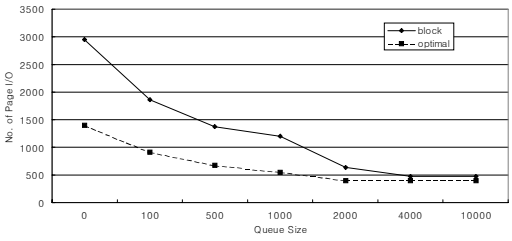


그림 5 성능 평가(큐의 크기 : 4,000)



(a) Q1



(b) Q3

그림 6 큐 크기에 따른 페이지 I/O (비 클러스터링 비율 : 20%)

면 비 클러스터링 비율이 증가하더라도 페이지 I/O는 크게 늘지 않는 것을 알 수 있다. 하지만 그림 5 (a), (c)에서 시그니처 방법은 탐색 공간을 아주 많이 줄이기 때문에 블럭 탐색보다 훨씬 나은 성능을 보인다는 것을 알 수 있다. 그 이유는 트리의 상단부에서 그래프의 탐색을 줄여서 탐색 공간이 많이 줄기 때문이다. 시그니처 방법이 트리의 탐색 공간을 많이 줄이지 않을 경우에는 그림 5(b)에서 보는 바와 같이 블럭 탐색이 시그니처 탐색보다 더 나은 성능을 보인다는 것을 알 수 있다. 하지만 최적화 방법은 모든 경우에 가장 나은 성능을 보인다. 즉 시그니처나 블럭 탐색보다 훨씬 더 많은 페이지 I/O를 줄일 수 있으며 XML 질의를 처리하는데 가장 적합한 방법이라는 것을 알 수 있다.

블럭 탐색은 이웃 객체를 관리하고 상태 집합을 저장하기 위하여 큐를 이용한다. 만약 큐의 크기가 아주 커다면 페이지 I/O의 수는 가장 작을 것이다. 본 논문에서는 실험에서 그림 6과 같이 큐의 크기를 변경해 가면서 페이지 I/O를 측정하였다. 그림 6에서 x 축은 큐의 크기이다. Q2, Q4, Q5, Q6의 그래프는 Q3과 유사하다. 그림 6에서 보는 바와 같이 큐의 크기가 커질수록 성능이 좋아진다. 하지만 성능 향상은 큐의 크기가 4,000개 정도 될 때 수렴하게 된다. 이것은 큐에 객체를 저장하기 위하여 아주 많은 메모리가 필요한 것은 아니라는 것을 보여준다.

6. 결 론

DOM 트리의 각 노드는 객체 저장소에 객체로서 저장된다. 이러한 저장소에 객체들이 클러스터링 되어 있으면 페이지 I/O가 줄 수 있다. 하지만 많은 갱신 연산 후에는 같은 페이지에 있던 객체들도 여러 페이지에 흩어지게 된다. 객체들이 클러스터링이 되어있지 않을 때 최악의 경우 객체 하나를 읽을 때마다 페이지 폴트가 발생한다. XML 질의의 특징은 정규 경로식이 있다는 것이다. 본 논문에서는 정규 경로식을 빠르게 처리하기 위하여 세가지 스캔 연산자를 제공하였다. 블럭 탐색을 통하여 같은 페이지에 저장되어 있는 이웃 객체를 먼저 처리함으로서 페이지 I/O를 줄였다. 객체들의 위치 정보는 객체 저장소의 OID 테이블을 이용하였다. 이 방법은 객체들이 클러스터링 되어있지 않을 때 페이지 I/O를 많이 줄일 수 있다. 또한 탐색 공간을 줄이기 위하여 시그니처 기반 탐색 방법을 개발하였다. 최적화 탐색 방법은 이러한 두 방법보다 더 나은 성능을 보였다. 최적화 탐색은 아주 많은 페이지 I/O를 줄였으며 이것은 XML 질의를 처리할 때 매우 유용하다.

참 고 문 헌

[1] Serge Abiteboul., Querying Semistructured Data, *International Conference on Database Theory*, January 1997.

[2] P. Buneman, Semistructured Data, *ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, May 1997.

[3] Jason McHugh, Serge Abiteboul, Roy Goldman, Dallan Quass, and Jennifer Widom, Lore: A Database Management System for Semistructured Data, *SIGMOD Record*, 26(3), 9 1997.

[4] eXcelon, An XML Data Server For Building Enterprise Web Applications, *http://www.odi.com/products/white_papers.html*, 1999.

[5] Gerald Huch, Ingo Macherius and Peter Fankhauser, PDOM: Lightweight Persistency Support for the Document Object Model, *OOPSLA*, November, 1999

[6] S. Abiteboul, D. Quass, J. McHugh, J. Widom, and J. Wiener, The Lorel Query Language for Semistructured Data, *International Journal on Digital Library*, 1(1), 4 1997.

[7] Alin Deutsch, Mary Fernandez, Daniela Florescu, Alon Levy, and Dan Suciu, XML-QL: A Query Language for XML, *htttp://www.w3.org/TR/NOTE-xml-ql*, August 1998.

[8] Peter Buneman, Susan Davidson, Gerd Hillebrand, and Dan Suciu, A Query Language and Optimization Techniques for Unstructured Data, *SIGMOD*, 1996.

[9] V. Christophides, S. Abiteboul, S. Cluet, and M. Scholl, From Structured Documents to Novel Query Facilities, *SIGMOD*, 1994.

[10] Elisa Bertino and Won Kim, Indexing Techniques for Queries on Nested Objects, *IEEE Transactions on Knowledge and Data Engineering*, 1(2), 1989.

[11] Tova Milo and Dan Suciu, Index Structures for Path Expressions, *ICDT*, 1999.

[12] Sangwon Park and Hyoung-Joo Kim, A New Query Processing Technique for XML Based on Signature, *DASFAA*, April 2001.

[13] 박상원, 김형주, 시그니처를 이용한 XML 질의 최적화 방법, *정보과학회 논문지(데이타베이스)*, 28(1), March 2001.

[14] R.G.G. Cattell and Douglas K. Barry, *The Object Database Standard: ODMG 2.0*, Morgan Kaufmann Publisher, Inc., 1997.

[15] M. Kifer, W. Kim, and Y. Sagiv, Querying Object-Oriented Databases, *SIGMOD*, 1992.

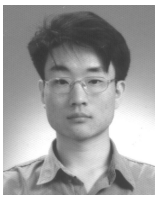
[16] Roy Goldman and Jennifer Widom, DataGuides: Enabling Query Formulation and Optimization in Semistructured Databases, *VLDB*, 1997.
[17] Jason McHugh and Jennifer Widom, Query Optimization for XML, *VLDB*, 1999.
[18] Alin Deutsch, Mary Fernandez, and Dan Suciu, Storing Semistructured Data with STORED, *SIGMOD*, 1999.
[19] Daniela Florescu and Donald Kossmann, Storing and Querying XML Data using an RDBMS, *Data Engineering Bulletin*, 22(3), September 1999.
[20] Jayavel Shanmugasundaram, Kristin Tufte, Gang He, Chun Zhang, David DeWitt, and Jeffrey Naughton, Relational Databases for Querying XML Documents: Limitations and Opportunities, *VLDB*, 1999.
[21] Takeyuki Shimura, Masatoshi Yoshikawa, and Shunsuke Uemura, Storage and Retrieval of XML Documents Using Object-Relational Databases, *DEXA*, 1999.
[22] Minos Garofalakis, Aristides Gionis, Rajeev Rastogi, S. Seshadri, and Kyuseok Shim, XTRACT: A System for Extracting Document Type Descriptors from XML Documents, *SIGMOD*, 2000.
[23] Hwan-Seung Yong, Sukho Lee, and Hyoung-Joo Kim, Applying Signatures for Forward Traversal Query Processing in Object-Oriented Databases, *ICDE*, 1994.
[24] Walter W. Chang and Hans J. Schek, A Signature Access Method for the Starburst Database System, *VLDB*, 1989.
[25] Peter Linz, *An Introduction to Formal Languages and Automata*, Houghton Mifflin Company, 1990.
[26] Won Kim, *Introduction to Object-Oriented Databases*, The MIT Press, 1990.
[27] Won Kim, A New Way to Compute the Product and Join of Relations, *SIGMOD*, 1980.

박 상 원

1994년 서울대학교 컴퓨터공학과(학사). 1997년 서울대학교 컴퓨터공학과(석사) 1997년 ~ 현재 서울대학교 컴퓨터공학부 박사과정. 관심분야는 데이타베이스, XML, Semistructured data, Web

박 동 주

1995년 서울대학교 컴퓨터공학과(학사). 1997년 서울대학교 컴퓨터공학과(석사). 2001년 서울대학교 컴퓨터공학과(박사). 2001년 ~ 현재 삼성전자. 관심분야는 데이타베이스, 멀티미디어 데이타베이스, 공간 데이타베이스



정 태 선

1995년 한국과학기술원 전산학과 학사(B.S.). 1997년 서울대학교 전산과학과 석사(M.S.). 1997년 ~ 현재 서울대학교 전산과학과 박사과정

김 형 주

1982년 서울대학교 전자계산학과(학사). 1985년 Univ. of Texas at Austin(석사). 1988년 Univ. of Texas at Austin (박사). 1988년 5월 ~ 1988년 9월 Univ. of Texas at Austin. Post-Doc. 1988년 9월 ~ 1990년 12월 Georgia Institute of Technology(부교수). 1991년 1월 ~ 현재 서울대학교 컴퓨터공학부 교수.