시그니처를 이용한 XML 질의 최적화
(XML Query Optimization based on Signature)

박 상 원 † 김 형 주 ††
(Sangwon Park)(Hyoung-Joo Kim)

요 약 XML은 웹에서 데이타를 주고받는 표준으로 새로이 등장하고 있다. 이러한 데이타를 잘 처리하기 위하여 데이타베이스의 도움은 필연적이다. XML을 처리하는 데이타베이스에서 데이타는 트리 형태로 저장되며 질의어는 정규 경로식(regular path expression)의 특징을 가지고 있다. 이때 질의 처리는 트리의 각 노드 탐색을 통하여 수행된다. 본 논문에서는 시그니처를 이용한 저장 방법 및 질의 처리를 통하여 질의 수행 시 각 노드의 탐색 횟수를 줄여 질의 수행을 빠르게 할 수 있게 한다. 뿐만 아니라 반 구조적 데이타에 대한 인덱스도 데이타베이스 내에서 반구조적 데이타로 표현된다. 그러므로 본 논문에서 제안한 방법을 인덱스 노드에도 적용할 수 있다. 이와 같이 본 논문에서 제안한 방법은 데이타 객체와 인덱스 객체의 탐색을 줄임으로서 XML 질의를 빠르게 처리할 수 있게 한다.

Abstract XML is an emerging standard for data representation and exchange on the Web. Database must support to manipulate XML data well. XML data is represented as tree structure and the query. is represented as regular path expression. This query is evaluated by traversing each node of the tree. In this paper we minimize the number of fetching the nodes when the XML query is evaluated using signature, which is used in both storing the XML objects and executing the queries. The structure of index of the semistructured data is represented as semistructured data also. Minimizing traverse of not only data object but also index object, we can execute the regular path expressions fast.

1. 서 론

웹에서 정보를 표현하고 주고받는 수단으로 XML이 새로운 표준으로 등장하고 있다. 앞으로 많은 정보가 XML로 표현되고 처리될 것으로 예상된다. 많은 XML 데이타를 효율적으로 다루기 위하여 데이타베이스의 도입은 필수적이다. 하지만 XML은 기존의 데이타 모델과는 다른 반구조적(semistructure)[1, 2]인 특징을 가지고 있다. 이러한 상이한 구조로 인하여 새로운 저장 방법 및 질의 처리 모델이 요구되고 있다.

데이타베이스 분야에서 연구되어 왔던 반구조적 데이타는 그래프 형태로 표현된다. 그러므로 반구조적 데이타 처리를 위한 연구 결과는 트리 형태로 표현되는 XML 데이타에도 대부분 그대로 적용될 수 있다[3]. 반구조적 데이타 혹은 XML 데이타를 처리하는 데이타베이스로 Lore[4], eXcelon[5] 등이 있다. 이러한 데이타베이스에서 그래프의 각 노드는 하나의 객체로서 저장되며 질의 처리기는 각 노드를 탐색하면서 처리된다. 이때 노드의 탐색 횟수를 줄이는 것이 질의 수행 속도를 빠르게 하는 관건이다.

XML-QL[6], XQL과 같은 XML 질의어는 객체지향형 데이타베이스의 질의어와 같은 경로식(path expression)을 지원한다. 이때 두 질의어의 가장 큰 차이점은 위 예제와 같이 정규 경로식(regular path expression)[7, 8, 9]을 지원하는 것이다.

```
SELECT x.company.(address|telephone)
FROM school.*.parent x;
```

위 질의는 학부모들의 회사 주소나 혹은 전화번호를 출력하는 것이다. 위 예제를 처리할 때 인덱스가 없을 경우 *로 인하여 그래프의 노드 대부분을 탐색해야 한다.

XML 문서는 트리 형태로 표현되며 XML 질의 처리기는 트리를 탐색하면서 질의를 처리한다. 본 논문에서는 트리의 각 노드에 해당 노드의 서브 트리에 대한 시그니처

· 이 논문은 2000년도 두뇌한국21 사업에 의하여 지원되었음.
† 비 회 원 : 서울대학교 컴퓨터공학부
swpark@oopsla.snu.ac.kr
†† 종신회원 : 서울대학교 컴퓨터공학부 교수
hjk@oopala.snu.ac.kr
논문접수 : 2000년 8월 16일
심사완료 : 2000년 12월 19일

(signature)[10, 11] 정보를 저장하는 s-DOM(signature based DOM) 저장 구조를 제안하였다. s-DOM은 트리의 한 노드가 그 노드의 서브 트리에 있는 모든 엘리먼트와 애트리뷰트 이름의 해쉬 값을 비트 연산 OR로 한 값을 가지고 있는 것이다. 즉 s-DOM은 하위 트리에 어떠한 이름을 가지는 엘리먼트 노드가 나오는지를 미리 유추하여 질의 처리시 그래프 탐색 범위를 줄일 수 있도록 하였다. 트리를 탐색하기 전에 노드 n의 서브 트리에 원하는 레이블 l을 가진 노드가 있는지 검사하기 위하여 노드 n의 시그니처 S_n와 찾고자 하는 레이블의 시그니처 S_l를 구하여 둘을 AND 연산한다. 그 결과가 S_l이면 노드 n의 서브 트리에 레이블 l을 가진 노드가 있을 가능성이 높다는 것을 나타내므로 서브 트리를 탐색하지만 그렇지 않을 경우 서브 트리에는 레이블 l을 가진 노드가 나타나지 않으므로 탐색할 필요가 없게 되는 것이다. 본 논문에서는 이와 같은 시그니처를 이용한 질의 처리를 위하여 s-NFA 방법을 제안하였다. s-NFA는 정규 경로식을 처리하는데 이것은 기존의 관계형 데이타베이스나 객체지향형 데이타베이스에서 제안하였던 시그니처 방법[10, 12]과 가장 큰 차이점이라 할 수 있다.

정규 경로식을 빠르게 처리하기 위하여 [3, 13, 14]와 같은 반구조적 데이타에 대한 새로운 인덱스 방법이 제안되었다. 이러한 인덱스들은 정규 경로식에 대한 오토마타와 익스텐트(extent)를 가지고 있어 주어진 정규 경로식과 일치하는 노드들을 빠르게 탐색할 수 있도록 도와준다. 이때 DataGuide[13]는 OEM[15]으로 표현되며 1-index, 2-index, T-index[14]도 역시 그래프로 표현된다. 즉 이러한 인덱스들은 그 자체가 다시 반구조적 데이타가 된다. 2-index와 같은 경우 최악의 경우 인덱스 노드의 개수가 데이타 노드 개수의 제곱배가 된다. 이와 같이 이러한 인덱스들은 노드의 개수가 상당히 많으며 정규 경로식 질의가 들어오면 인덱스 노드들을 탐색해야 한다. 그러므로 인덱스의 노드 탐색 횟수를 줄이는 것도 질의를 빠르게 수행하는 한 방법이 된다. 본 논문에서 제안한 시그니처 방법은 이러한 인덱스 노드에도 적용할 수 있으며 그 결과 인덱스 노드의 탐색 횟수도 줄일 수 있다.

s-DOM은 트리의 각 노드에 시그니처를 저장하므로 노드의 크기가 시그니처의 크기만큼 커진다. 하지만 그 크기가 몇 바이트 밖에 되지 않으므로 성능에 큰 영향을 미치지 않는다. 또한 연산도 비트 연산이므로 시그니처 비교로 인한 오버헤드도 거의 없다. 이러한 방법은 인덱스를 사용할 수 없는 경우에 아주 유용하다. 그 뿐만 아니라 반구조적 데이타에 대한 인덱스의 각 노드에도 본 논문의 기법을 적용할 수 있으며 이는 인덱스 노드 탐색 횟수를 줄인다.

본 논문에서 제안한 방법은 XWEET(XML DBMS for Web Environment)[16] 시스템의 질의 최적화 모듈에서 사용하는 방법이다. XWEET는 XML 데이타의 효율적인 저장, 추출, 질의 및 웹 환경에서의 응용을 위한 기반 시스템이다. 이것은 XML 데이타의 저장을 위한 PDM(Persistent Data Manager), 소스로부터 데이타 통합을 위한 랩퍼와 XWS(XWEET Web Wrapper System) 및 웹 응용 프로그램을 작성하기 위한 WPG(Web Page Generator), HTML/XML Template등을 제공한다.

본 논문의 구성은 다음과 같다. 2장에서는 관련연구에 대해서 설명하고 3장은 데이타 모델 및 질의어에 대하여 설명한다. 4장은 시그니처를 적용한 XML 객체에 대하여 설명하고 5장은 시그니처를 이용한 질의 최적화 방법에 대하여 설명한다. 6장에서는 시그니처를 이용한 것과 이용하지 않은 것에 대한 성능을 비교하며 마지막으로 결론 및 향후 연구 방향에 대하여 설명한다.

2. 관련 연구

시그니처 방법은 정보 추출 및 데이타베이스 분야에서 많이 연구되어 왔다[10, 11, 12, 17]. 시그니처 방법은 필요한 단어들의 해쉬 값을 구하여 간단한 비트 연산만으로 원하는 데이타를 빨리 찾기 위하여 사용되었다. [10]에서는 관계형 데이타베이스에서 B 트리를 사용하지 않고, 특정 애트리뷰트의 값들에 대한 시그니처를 구한 후 간단한 비트 연산만으로 원하는 애트리뷰트를 가진 튜플을 찾을 수 있게 하였다. [12]에서는 객체지향형 데이타베이스에서 OID로 참조하는 객체의 특정 애트리뷰트의 시그니처를 구하여 OID 뿐만 아니라 피참조 객체의 객체 시그니처도 같이 저장하였다. 이때 OID가 가리키는 객체를 참조하기 전에 객체 시그니처를 먼저 비교함으로서 객체를 읽어들이는 연산을 줄일 수 있게 하였다.

관계형 데이타베이스 혹은 객체지향형 데이타베이스에서 사용한 시그니처 방법은 특정 애트리뷰트를 가진 객체를 찾을 때 사용되었다. 본 논문에서 제안한 방법은 이와는 다르게 XML 문서의 엘리먼트와 애트리뷰트 이름에 대하여 시그니처를 구한 후 이를 상위 노드로 전파하여 임의의 하위 노드에 특정 이름을 가진 노드가 있는지를 찾아내는 것이 차이점이라 할 수 있다. 또한 [12]에서 처리하지 못했던 정규 경로식을 가진 질의를 처리할 수 있는 방법을 제안한 것이 가장 큰 차이점이라 할 수 있다.

XML 문서를 기존의 데이타베이스에 효율적으로 저장하기 위한 방법은 활발히 연구되고 있다[18, 19, 20, 21]. 이런 방법들은 관계형 데이타베이스에 XML 문서를 저장하기 위한 스키마 생성 방법과 질의 변환을 통한 데이타 추출에 중점을 두고 있다. 기존의 데이타베이스에 XML 문서를 저

장하는 것은 크게 두가지 방법으로 나누어 볼 수 있다. [18, 20]에서와 같이 XML 문서의 태그를 이용하여 관계형 데이타베이스 스키마를 생성한 후 각각의 엘리먼트를 대응하는 적절한 테이블에 저장하는 방법과 [19, 21]와 같이 XML 데이타의 트리 구조를 그대로 유지하는 방법이다.

트리의 각 노드를 객체 단위로 저장하는 것으로 eXcelon [5], PDOM[22] 등이 있는데 이것은 각각의 노드를 하나의 객체로 저장하기 때문에 XML 문서의 구조가 변하지 않으며 원래 문서의 구조를 그대로 유지할 수 있다. 객체지향 데이타베이스나 Lore[7] 등에서도 이와 같은 방법을 사용한다. 본 논문에서는 이와 같은 방법으로 객체를 저장한다고 가정하고 있다. 이때 객체의 접근 횟수를 줄이는 것이 질의를 최적화할 수 있는 한 방법이다.

XML 데이타에 대한 질의로는 정규 경로식[7, 8, 9]을 이용한다. 이러한 정규 경로식을 빠르게 수행하기 위하여 [3, 13, 14]와 같은 인덱스가 제안되었다. 이것은 익스텐트와 오토마타가 같이 있는 구조로서 정규식에 해당하는 것을 오토마타를 통해 찾아서 익스텐트를 반환하는 구조로 되어 있다.

XML 문서의 자료 구조는 DOM[23]으로 표현하며 이것은 트리 형태의 그래프 구조로 되어 있다. DOM 인터페이스는 한 노드에서 그 노드의 부모, 자식, 형제 노드들을 탐색할 수 있는 인터페이스를 제공한다. 이러한 인터페이스를 통하여 트리를 탐색하게 된다.

3. 데이타 모델과 질의어

본 논문에서 다루는 데이타는 XML이다. 이것은 반구조적 데이타 모델인 OEM[15]과 유사하다. OEM은 그래프 형태로 표현되지만 XML은 트리 형태의 구조를 가진다. XML 데이타 구조에 대한 인터페이스로 DOM[23]이 제안되어 있다. 이것은 트리 형태로 이루어진 XML 데이타 접근에 대한 표준 인터페이스로서 본 논문에서 다루는 데이타 모델이다. DOM은 노드들이 트리 형태로 이루어진 구조로서 각 노드는 부모, 자식, 형제 노드들을 가리키고 있다. 이때 노드들은 순서를 가지는 리스트이다.

그림 1은 XML 파일의 예이다. 이것은 DTD가 없는 XML 문서로서 XML 문서는 DTD를 가질 수도 있고 DTD가 없을 수도 있다. DTD는 문서의 정확성(validity)을 확인하는데 사용되며 본 논문에서는 DTD 정보를 이용하지 않는다. 이 XML 문서를 DOM 구조로 표현한 것이 그림 2이다. 이것은 트리 형태의 구조를 가지며 각 노드에서 부모, 자식, 형제 노드를 탐색할 수 있다.

XML 질의는 이와 같은 트리 형태의 그래프의 각 노드를 탐색하면서 수행된다. 그러므로 이 노드의 탐색 횟수를

```
<?xml version="1.0">
<!DOCTYPE AddrList>
<AddrList>
  <person>
    <name>
      <first>John</first>
      <last>Smith</last>
    </name>
    <company>IBM</company>
  </person>
  <person name="Robert Johnson">
    <company>
<address>Heidelberg</address>
<telephone>123-4567</telephone>
    </company>
    <father>
      <person>
        <name>William
Johnson</name>
      </person>
  </person>
  <company>
    <name>Samsung</name>
    <address>Suwon</address>
<telephone>549-0987</telephone>
  </company>
</AddrList>
```

그림 1 XML 파일의 예

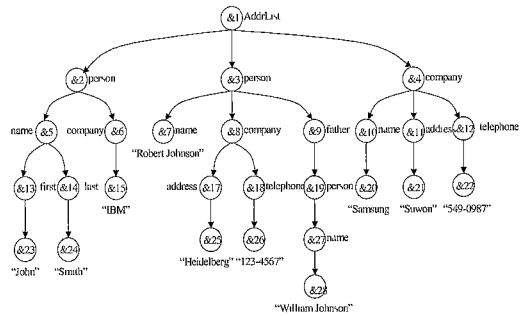


그림 2 DOM 그래프

줄이는 것이 XML 질의 처리의 핵심이다. 본 논문에서 각각의 노드는 객체로 저장하며 각각의 OID는 그림 2에서와 같이 &를 붙여 표시한다. 예를 들어 루트 노드의 OID는 &1 이다.

본 논문에서 다루는 질의는 정규 경로식을 가진 질의이다. 주어진 정규 경로식을 가지는 질의를 탐색할 수 있게 하는 스캔 인터페이스를 제공하며, 이 인터페이스를 통하여 정규 경로식을 만족하는 노드를 가져온다. 각각의 노드가 클러스트링 되지 않고 디스크에 저장되었다면 노드를 하나 읽어들이는 것은 디스크 페이지 하나를 읽어들이는 것이다. 그러므로 디스크 I/O를 줄이려면 노드의 탐색 횟수를 줄여야 한다. 정규 경로식을 처리하기 위하여 노드를 탐색할 때 노드 탐색 횟수를 줄이는 것이 본 논문의 목적이다.

4. 시그니처(Signature) 기법을 적용한 XML 문서의 저장

이번 절에서는 시그니처를 이용한 질의 처리를 위하여 XML 문서에 시그니처 기법을 적용하는 방법에 대하여

Tom	01001000
John	00100001
Kennedy	10001000

그림 3 시그니처의 예

설명하고 다음절에서는 시그니처를 이용하여 저장된 XML 문서에 대한 질의 처리에 대하여 설명한다.

4.1 시그니처란?

시그니처[10, 11, 17]는 일반적인 텍스트 문서에서 검색을 빠르게 하기 위한 방법으로 제안되었다. 시그니처는 문자열에 대한 해쉬값으로서 본 논문에서는 [11]의 SC 방법으로 시그니처를 만들었다. 예를 들어 그림 3에서와 같이 각각의 문자열에 대한 시그니처를 구했다고 하자. 각각의 문자열이 한 문서 블럭 D에 저장되어 있다면 그 문서 블록 D의 시그니처 S_D는 그 블럭 내의 모든 시그니처의 OR 연산으로 구한다. 그러므로

S_D = 11101001 = (01001000 | 00100001 | 10001000)

이다. 이때 이 페이지 내에 문자열 "Tom"이 있는지를 알아보려면 아래 조건을 만족하면 된다.

$S_{"Tom"} \equiv S_{"Tom"} \wedge S_D$

즉 Tom의 시그니처 $S_{"Tom"}$(01001000)과 블럭 D의 시그니처 S_D(11101001)에 대하여 비트 연산 AND를 했을 때 그 결과가 $S_{"Tom"}$이 되면 그 블럭 내에 Tom이 있을 가능성이 높은 것이다. 이와 다르게 Jane의 시그니처 $S_{"Jane"}$이 01010001 이라면 $S_D \wedge S_{"Jane"}$의 결과가 $S_{"Jane"}$이 되지 못하므로 그 블럭 내에는 Jane이 없다는 것을 확신할 수 있다. 이와 같이 특정 단어가 들어간 문서를 찾을 때 그 단어의 시그니처와 문서 블럭의 시그니처를 AND 연산함으로써 검색 연산의 비용을 줄일 수 있다.

시그니처는 여러 블럭으로 저장된 문서에서 특정 문자열을 찾을 때 모든 문서 블럭을 검색하지 않고 문서 블럭에 대한 시그니처를 이용하여 시그니처가 매칭되는 블럭만 검사하게 하여 원하는 결과를 빠르게 찾을 수 있게 한다. 이와 같은 시그니처 기법을 DOM에 적용한 것이 다음절에 나오는 s-DOM이다.

4.2 s-DOM (Signature based DOM)

본 논문에서 제안하고 있는 시그니처 방법은 XML 데이타를 저장할 때 그림 2에서 보는 바와 같은 그래프의 각 노드를 객체 단위로 저장한다고 가정한다. 이것은 반구조적 데이타를 위한 데이타베이스[4]나 기존 데이터베이스[5, 19, 21]에서 실제 사용하고 있는 저장 방법이다. 여기서 DOM의 각 노드를 저장할 때 시그니처 정보를 첨가하여 저장하게 된다. 시그니처는 각 엘리먼트와 애트리뷰트 이름을 이용하여 구하고 상위 노드는 하위 노드의 시그니처 정보를 포함하고 있는 형태이다. 먼저 각각의 엘리먼트와 애트리뷰트 이름의 해쉬 값을 구하면 표 1의 왼쪽과 같다. 즉 company의 해쉬 값은 이진값으로 00001001 이다. 이 해쉬 값을 이용하여 DOM에서 시그니처 정보를 더한 s-DOM을 구축한다. s-DOM은 하위 노드의 시그니처 값을 OR 연산하여 구한다. 즉 하위 노드의 시그니처의 OR 값을 상위 노드의 시그니처로 갖는다. 이것은 노드의 이름으로 시그니처를 만드므로 엘리먼트 노드나 애트리뷰트 노드만 시그니처 값을 구하고 텍스트 노드는 시그니처 값을 구하지 않는다. 알고리즘 1은 특정 노드의 시그니처를 구하는 방법으로 이렇게 구한 각 노드 객체의 시그니처 값은 표 1의 오른쪽과 같다.

표 1 각 엘리먼트 이름의 해쉬 값과 각 노드의 시그니처 값

AddrList	01001000	person	00100010
name	10001000	first	10100000
last	01000010	company	00001001
address	01000001	telephone	00101000
father	00000011		

&1	11101011	&2	11101011	&3	11101011
&4	11101001	&5	11100010	&6	00000000
&7	00000000	&8	01101001	&9	10101010
&10	00000000	&11	00000000	&12	00000000
&13	00000000	&14	00000000	&17	00000000
&18	00000000	&19	10001000	&27	00000000

예제 4.1 (시그니처 계산) 그림 2의 &2 객체의 시그니처를 구하면 다음과 같다. 먼저 자식 노드 &5, &6의 시그니처를 구해야 한다. 이때 노드 &6, &13, &14의 시그니처는 자식 노드가 텍스트 노드이므로 시그니처가 0이다. 노드 &5의 시그니처는 &13, &14의 시그니처와 노드 &13, &14의 이름의 해쉬 값을 OR 연산하므로 11100010이다. 그러므로 &2의 시그니처는 &5, &6의 시그니처 11100010, 0과 &5, &6의 노드 이름 name, company의 해쉬 값을 OR 연산한 11101011이다.

예제 4.1에서 보는 바와 같이 상위 노드는 하위 노드의 시그니처 정보를 가지고 있다. 이것이 의미하는 바는 시그니처 정보를 보면 하위 노드에 어떤 이름을 가진 노드가 나타나지 않는지, 혹은 어떤 노드 이름이 나타날 가능성이 높은지 알 수 있다. 예제 4.2는 시그니처를 이용하여 특정 레이블을 가진 노드가 있는지검사하는 방법에 대하여 설명한다.

```
알고리즘 1 MakeSignature(node)
s ← 0
if node is an Element or Attribute node then
  for each ChildNode of node do
    s ← s ∨ MakeSignature(ChildNode)  /* 비트 연산 */
    s ← s ∨ Hash(ChildNode.Name)  /* 비트 연산 */
  end for
end if
node.signature ← s
```

예제 4.2 (노드 탐색) 그림 2의 &3 노드의 하위 노드에 address 노드가 나타날 가능성이 있는지 검사하는 방법은 address의 해쉬값 $S_{"address"}$와 &3 노드의 시그니처 $S_{\&3}$를 비트 연산 AND 하여 그 결과로 $S_{"address"}$가 나오면 나타날 가능성이 높은 것이다. 즉, $S_{"address"} \wedge S_{\&3} \equiv S_{"address"}$와 같은 결과가 나오면 &3의 하위 노드에서 address가 나올 가능성이 있다는 얘기이다. 이와 다르게 father가 &4 객체 아래에 있는지 알아보면 $S_{"father"} \wedge S_{\&4} \neq S_{"father"}$이다. 그러므로 &4 객체 아래에는 father 이름을 가지는 노드가 없다는 것을 알 수 있다. 즉 s-DOM은 하위 노드에서 나타나는 엘리먼트의 시그니처를 가지고 있음으로서 질의 처리시 하위 노드의 특정 엘리먼트가 나타나지 않는 그래프를 탐색 범위에서 제외할 수 있다.

5. s-NFA(Signature based NFA)

XML 질의어의 특징은 질의에서 정규 경로식을 사용할 수 있다는 것이다. 이를 처리하기 위하여 본 논문에서는 질의로 들어온 정규 경로식을 NFA로 변경한 후, 각 노드의 탐색을 NFA의 상태 변화로 반영하여 NFA의 상태가 최종 상태(final state)에 오면 그 노드가 질의를 만족하는 것으로 처리하는 방식을 취한다. 5.1절에서는 일반적인 정규 경로식을 NFA를 이용하여 처리하는 방법에 대하여 설명하고, 5.2절에서는 정규 경로식을 s-NFA로 변환한 후 시그니처 정보를 이용한 질의 처리에 대해서 설명한다.

5.1 NFA를 이용한 질의 처리

정규식은 오토마타를 이용하여 표현할 수 있다. 이것은 결정적 오토마타(deterministic automata)와 비결정적 오토마타(non-deterministic automata, NFA)로 나타낼 수 있다[24]. 그러므로 정규 경로식은 NFA로 표현할 수 있다. 이때 아무리 복잡한 NFA도 기본적으로 그림 4의 $L(r_1)L(r_2)$, $L(r_1 + r_2)$, $L(r*)$와 같은 세개의 조합 방법으로 표현할 수 있다[24]. 이때 $L(r?)$, $L(r+)$는 $L(r*)$의 변형으로 모두 표현될 수 있다. 즉 모든 정규 경로식은 NFA로 표현할 수 있으며 질의의 처리는 데이타 그래프의 각 노드 탐색에 따라 NFA의 상태를 전이하면서 그 상태가 최종 상태에 도달하게 되면 정규 경로식에서 요구하는 노드라는 것을 알 수 있게 되는 것이다. 이와 같은 정규 경로식은 다음과 같은 규칙에 의해 만들어진다.

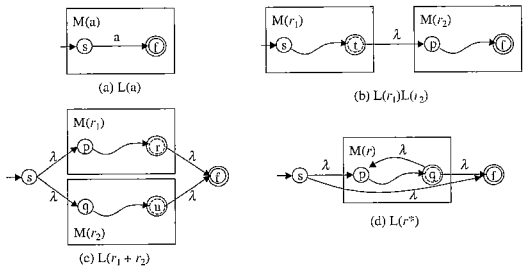


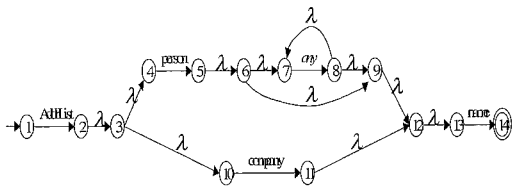
그림 4 NFA

• 원자값의 NFA는 그림 4(a)와 같이 시작 상태(start state)와 최종 상태(final state) 하나씩을 가지며 시작 상태에서 최종 상태로 원자값을 레이블로 하는 에지(edge)가 하나 있다.

• 각각의 NFA는 그림 4의 (b), (c), (d)와 같은 방법으로 만들어지며 그 자체가 다시 시작 상태와 최종 상태 하나씩을 가진 새로운 NFA가 된다.

이와 같은 연산을 통하여 시작 상태 하나와 최종 상태 하나가 있는 정규 경로식에 대한 NFA가 만들어진다.

예제 5.1 정규 경로식 AddrList.((person.*)|company).name에 대한 NFA는 위 규칙에 의해 아래 그림과 같이 만들어진다. 이때 *는 어떠한 레이블도 가능하므로 (*any label*)*와 같다. 각각의 NFA가 합해져서 하나의 NFA를 구성하게 되며 시작 상태와 최종 상태를 하나씩 가지게 된다.



위 예제와 같은 NFA는 질의 처리기에서 사용된다. 예제 5.2에서 NFA를 이용한 정규 경로식의 처리에 대하여 설명한다.

예제 5.2 정규 경로식 AddrList.((person.*)|company).name을 그림 2에서 수행하면 그 결과는 {&5, &7, &27, &10}이 차례로 반환된다. 이것은 깊이 우선 탐색(depth first search)[1]으로 DOM 트리의 각 노드를 방문하면서 NFA를 만족하는 노드를 찾은 것이다. 이때 person.*로

인하여 person 노드 아래의 노드는 모두 방문해야 한다.

5.2 s-NFA

5.1절의 NFA에서 특정 상태에서 다른 상태로 전이하는 것은 에지(edge)의 레이블만 가지고 결정한다. 상태 전이를 통하여 최종 상태에 도달하면 그 NFA를 만족하는 것으로 결정한다. 이와 같은 NFA에서는 특정 상태에서 뒤에 어떠한 노드를 거쳐야만 최종 상태로 전이할 수 있는지 알 수 없고, 더 이상 상태 전이를 할 수 없을 때까지 계속 상태 전이를 적용해 나가는 방법밖에는 다른 방법이 없다. s-NFA는 본 논문에서 제안하는 질의 처리를 하기 위하여 NFA에 시그니처 정보를 더한 것으로서 이것은 NFA의 각 노드에 NFA의 최종 상태에 도달하기 위하여 반드시 거쳐야 하는 에지들의 레이블에 대한 시그니처 정보를 가지고 있는 것이다. s-NFA의 시그니처와 s-DOM의 시그니처를 이용하여 질의를 처리하게 되는데 s-DOM에서는 어떠한 레이블을 가진 노드가 서브 트리에 존재하는지를 나타내는 것이고, s-NFA의 시그니처는 질의를 만족하기 위해서 반드시 나타나야 하는 레이블들을 나타낸다. 그러므로 s-NFA의 시그니처와 s-DOM의 시그니처를 비교하여 s-DOM의 서브 트리를 검색할지 여부를 결정하게 된다. 즉 s-NFA에서 상태 전이를 통하여 거부(reject)되지 아니하더라도 s-NFA의 시그니처가 s-DOM의 시그니처에 나타나지 않으면 s-NFA에서 더 이상 상태 전이를 할 수 없으므로 서브 트리를 방문하지 않는 것이다. 아래 정의는 본 논문에서 유용하게 쓰이는 것들이다.

정의 5.1 (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 5.2 (경로 시그니처(path signature)) NFA의 한 상태 노드 n의 경로 시그니처 PS_n은 다음과 같이 정의된다.

$PS_n = \{x \mid x$는 NFA의 상태 노드 n의 한 NFA 경로에 나타나는 모든 레이블의 시그니처 값을 비트 연산 OR한 값$\}$

경로 시그니처는 NFA의 상태 노드에서 그 오토마타를 승인(accept)하기 위하여 반드시 나타나야 하는 레이블들에 대한 정보를 비트 값으로 가지고 있는 것을 말한다. NFA의 특정 상태에서 최종 상태로 갈 수 있는 길은 여러 개이기 때문에 이런 경로 시그니처는 집합으로 표현되며 NFA 상태 노드 n의 경로 시그니처의 집합은 PS_n으로 표현된다. 그러므로 s-NFA는 각 상태 노드가 경로 시그니처를 가진 NFA를 말한다.

그러면 경로 시그니처를 생성하는 방법에 대하여 알아보자. 그림 4는 모든 종류의 NFA를 만드는 방법에 대하여 나타낸 것이다. 즉 이들 그래프의 경로 시그니처를 구하는 방법은 모든 NFA의 경로 시그니처를 구하는 것과 같다. 다음은 그림 4의 각 그래프에서 경로 시그니처를 구하는 방법에 대한 규칙들이다.

규칙 5.1 (L(a)) 그림 4(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 다음과 같다.

$$PS_s = \{ H_a \}$$
$$PS_f = \{ 0 \}$$

규칙 5.2 (L(r_1 + r_2)) 그림 4(c)와 같이 두 개의 NFA가 |로 연결된 정규식인 경우 PS_s, PS_f는 다음과 같다.

$$PS_s = \{ x \mid x\text{는 } PS_p \text{ 혹은 } PS_q\text{의 한 원소}\}$$
$$PS_f = \{ 0 \}$$

규칙 5.3 (L(r*)) 그림 4(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.

$$PS_s = \{ 0 \}$$
$$PS_f = \{ 0 \}$$

규칙 5.4 (L(r+)) L(r+)는 그림 4(d)에서 s노드에서 f 노드로 가는 에지를 삭제하면 된다. 이때 PS_s 노드를 제외하고는 규칙 5.3과 동일하다.

$$PS_s = PS_p$$

L(r?)는 규칙 5.3과 동일하다.

규칙 5.5 (L(r_1)L(r_2)) 두 개의 NFA가 합해지는(concatenation) 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 n의 경로 시그니처 PS_n은 다음과 같다.

$PS_n = \{ (x \vee y) \mid x$는 원래의 경로 시그니처 PS_n의 한 원소, y는 경로 시그니처 PS_p의 한 원소 $\}$

1) 넓이 우선 탐색으로 트리를 탐색할 수도 있으나 이 경우 형제(sibling) 노드를 모두 탐색할 때 다시 방문하기 위하여 모두 큐에 삽입해 두어야 한다. 이 경우 데이타베이스의 형제 노드가 몇 개일지 예측할 수 없으므로 큐의 크기가 아주 커질 수 있다. 그러므로 넓이 우선 탐색은 XML 질의를 처리하는 방법으로 적절하지 않다.

1	{11101010, 11001001}	2	{10101010, 10001001}	3	{10101010, 10001001}
4	{10101010}	5	{10001000}	6	{10001000}
7	{10001000}	8	{10001000}	9	{10001000}
10	{10001001}	11	{10001000}	12	{10001000}
13	{10001000}	14	{00000000}		

예제 5.3 (NFA에서의 경로 시그니처) 위에서 정의한 규칙에 따라 예제 5.2의 경로 시그니처를 구하면 다음과 같다.

예를 들어 13번 노드는 반드시 name 에지를 지나야만 최종 상태로 갈 수 있다. 그러므로 13번 노드의 경로 시그니처 PS_{13}은 표 1에서 name의 해쉬값인 {10001000}을 가진다. 최종 상태인 14번 노드에는 더 이상 지나가야 할 에지가 없으므로 PS_{14}로 {00000000}을 가진다. 이때 11번 노드는 name을 지나야 하므로 PS_{11}은 PS_{13}과 같은 시그니처 값을 가지고, PS_{10}은 company와 name을 지나야 하므로 company와 name의 시그니처를 OR 연산한 값인 {10001001}을 가진다.

5.3 s-NFA를 이용한 질의 처리

s-NFA를 이용하여 질의를 처리하는 과정을 살펴보면 다음과 같다. s-NFA의 경로 시그니처는 오토마타에서 상태를 변이하기 위하여 반드시 거쳐야 하는 레이블이 어떠한 것인지를 알려주는 값이고, s-DOM에서의 각 노드

알고리즘 3 ForwardLambda(*SS*, *node*)

```
for each state node n which can go forward by λ in SS do
  for each signature S_i of PS_n do
    if S_i ∧ node.signature ≡ S_i then
      m ← the state node moved from n by λ
      add m to SS
      break
    end if
  end for
  remove n from SS
end for
```

의 시그니처는 하위 트리에 어떠한 레이블을 가진 노드들이 있는 지를 알려주는 값이다. 알고리즘 2는 스캔 인터페이스로서 정규 경로식에 해당하는 노드를 반환하는 함수이다. 이 함수는 ForwardLabel과 ForwardLambda를 호출한다. ForwardLabel은 s-NFA에서 주어진 레이블을 이용하여 상태를 전이하는 것이다. 알고리즘 3의 함수 ForwardLambda은 s-NFA 상태를 전진할 때 s-NFA와 s-DOM의 시그니처를 비교 연산하여 다음 상태로 전진할 수 있는지를 판단한다. 알고리즘 3의 if 절의 의미는 s-NFA의 최종 상태까지 전진하는데 필요한 레이블이 s-DOM의 서브 트리에 있는 지를 확인하는 것이다. 만약 없다면 상태 전진을 할 수 없으므로 그 노드는 탐색할 필요가 없게 되는 것이다.

알고리즘 2 next( )

```
/* SS is the state set of s-NFA */
node ← get next node by DFS from s-DOM
while node is not NULL do
  ForwardLabel(SS, node)
  ForwardLambda(SS, node)  /* 시그니처 이용 */
  if there is a final state in SS then
    return node
  end if
  if SS is empty then
    node ← get next node by DFS from s-DOM
  end if
end while
```

예제 5.4 (질의 처리) 예제 5.2의 질의를 처리할 때 이 질의를 s-NFA로 변환하면 s-NFA는 예제 5.2의 NFA 그림에 예제 5.3의 시그니처를 각 노드가 가지고 있는 형태가 된다. 이것을 그림 2의 s-DOM에 적용하였을 때 &1 노드를 읽으면 s-NFA에서의 상태 집합 S = {2}가 된다. 이때 λ로 된 상태를 전진하기 위하여 알고리즘 3를 적용하면 상태 노드 2의 시그니처 {10101010, 10001001} 중 10001001과 &2의 시그니처 11101011을 AND 연산하면 10001001이 되므로 상태 전이를 할 수 있게 되어 S = {3}이 된다. &2 노드를 이용하여 같은 연산을 수행하면 S = {7, 13}이 된다. 이때 상태 노드 7의 시그니처 10001000과 &5의 시그니처 11100010과 AND 연산을 하면 그 결과가 10001000이 되지 않는다. 그러므로 질의 person.*에도 불구하고 &5의 서브 트리는 방문할 필요가 없게 된다. 이와 같은 연산을 반복적으로 하면 원하는 결과를 얻을 수 있다.

삽입과 삭제 연산

삽입과 삭제 연산이 일어나면 그 부모 노드들의 시그니처 값을 변경하여야 한다. 한 노드의 갱신 연산이 일어나면 그 노드의 부모 노드의 시그니처 값은 부모 노드의 모든 자식 노드들의 시그니처를 OR 연산하여 다시 만들어야 한다. 이것은 부모 노드의 갱신 작업이 되므로 다시 재귀적으로 이와 같은 연산이 최상위 노드까지 연산되어야 한다. 이러한 점은 시그니처 방법을 사용하지 않았을 경우보다 갱신 연산시 비용이 많이 든다. 본 논문에서는 검색 연산의 수행 속도를 증진시키는 것이 목적이기 때문에 갱신 연산 시의 이와 같은 비용 상승은 고려하지 않았다. 일반적으로 XML 문서에서 갱신 연산은 자주 일어나는 것이 아니며 주로 검색 연산을 하는 경향이 있으므로 이와

같은 가정은 타당하다고 본다. 알고리즘 4는 갱신 연산이 일어날 때 DOM의 각 노드의 시그니처를 변경하는 알고리즘이다.

알고리즘 4 PropagateSignatureToParentNode(*node*)

```
CurrentNode ← node
while CurrentNode is not null do
  Signature ← 0
  for each ChildNode of CurrentNode do
    Signature ← Signature ∨ ChildNode.Signature /*bitwise 연산 */
  end for
  CurrentNode.Signature ← Signature
  CurrentNode ← parent node of CurrentNode
end while
```

6. 실험 결과

본 논문의 실험은 Java로 코딩되었으며 입력된 정규 경로식을 처리하는 스캔 연산자를 수행할 때 DOM에서 접근하는 노드 개수를 측정하였다. 또한 노드들을 깊이 우선 탐색(DFS) 방법과 넓이 우선 방식(BFS)으로 저장하였을 때 접근되는 디스크 페이지의 개수를 비교하였다. 디스크 접근은 페이지를 읽어들이는 횟수와 디스크 연산 시 소요되는 총 시간을 비교하였다. 이때 버퍼는 LRU 방식으로 페이지 치환(replace)을 하며 디스크 연산 시간을 계산하는 식은 [25]를 이용하였으며 각 인자는 표 2에 나타나 있다. 이때 Seek Time 식의 d는 트랙과 트랙사이의 거리를 나타낸다.

디스크에 저장하는 각 노드의 시그니처 크기는 5 바이트로 하였으며 버퍼의 개수는 20개로 고정하였다. 페이지의 크기는 4 Kbytes로 하였으며 각 페이지에 저장되는 객체의 크기는 엘리먼트의 이름에 따라 가변적으로 하였다.

표 2 디스크 연산 시 인자

섹터 크기	256 byte	실린더	1149
실린더 당 트랙 수	8	트랙 당 데이타 섹터 수	113
회전 속도	4002 RPM	콘트롤러 오버헤드	1.1 ms
Seek Time	$3.45+0.597\sqrt{d}$		

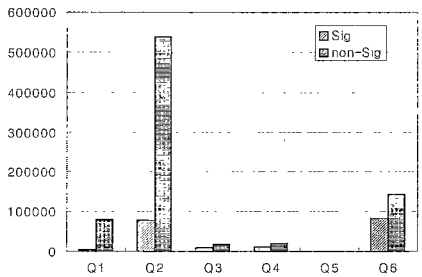
표 3 실험에 쓰인 질의

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

본 논문에서 디스크에 저장하는 방법으로 BFS와 DFS를 이용한 것은 트리 형태의 데이타를 저장하는 방법에서 가장 클러스트링이 많이 된 두 가지 형태를 택하였기 때문이다. 노드의 접근 횟수를 비교하는 것과, BFS, DFS로 저장한 후 페이지 I/O 횟수를 비교하는 것은 양극단을 비교하는 것으로, 전자는 클러스트링이 전혀 되어 있지 않은 상태를 나타낸 것이며, 후자는 완전히 클러스트링이 되어 있는 것을 말한다. 노드 접근 횟수는 결국 객체의 접근 횟수와 동일하며 실제 삽입과 삭제 연산이 빈번히 일어나게 되면 객체들의 저장 형태는 클러스트링이 되어있는 상태와 완전히 되어있지 않은 상태의 중간 형태를 띄게 될 것이다. 본 논문에서는 시그니처 방법을 이용하여 데이타를 저장할 때 클러스트링 효과를 보기 위하여 어떠한 방법을 이용하는 것이 좋은 지 실험을 통하여 비교하였다.

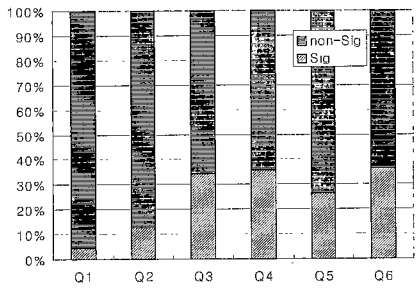
본 논문에서 실험한 데이타는 Shakespeare, The Book of Mormon과 Michael Ley의 논문 리스트 일부를 XML로 변환한 데이타[2]이다. 실험의 결과는 그림 5에 나타나있다. (a), (c), (e)는 각각 노드의 탐색 횟수, 디스크 I/O 횟수 및 디스크 연산에 걸린 시간을 측정한 것이다. (b), (d), (f)는 각 연산에서 사용한 방법들의 상대적인 값을 비교한 것이다. 여기에서 쓰인 질의는 6가지인데 표 3에 나타나 있다. 여기서 *[2]는 임의의 경로가 두개 있다는 것을 나타낸다. 보는 바와 같이 한 문서에 대하여 첫 번째 질의는 특정 경로에 있는 데이타를 검색하는 질의이고 두 번째 질의는 트리의 깊이와는 관계없이 모든 특정 레이블이 나타나는 것을 검색하는 질의이다.

그림 5 (a), (b)를 보면 각 질의를 수행할 때 노드의 방문 횟수를 알 수 있다. 시그니처 기반의 질의 처리가 모든 경우에 성능이 좋은 것을 알 수 있다. 이것은 시그니처를 이용하여 그래프 탐색을 줄일 수 있기 때문이다. 객체지향형

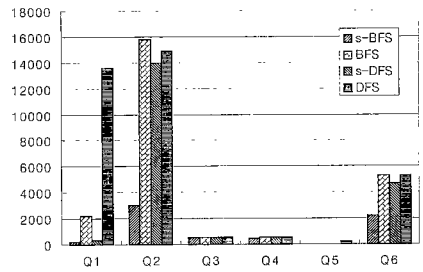


(a) 노드 탐색 횟수

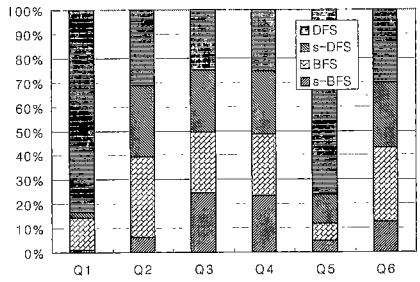
2) 각 데이타의 크기는 Shakespeare는 7.5 Mbytes, The Book of Mormon은 6.7 Mbytes, Bibliography는 247 Kbytes이다.



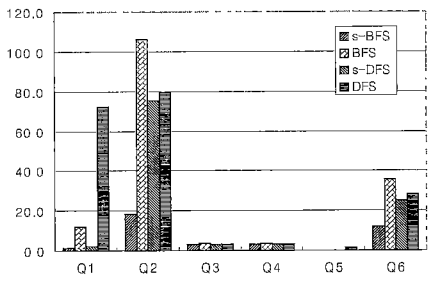
(b) 노드 탐색 횟수 비율



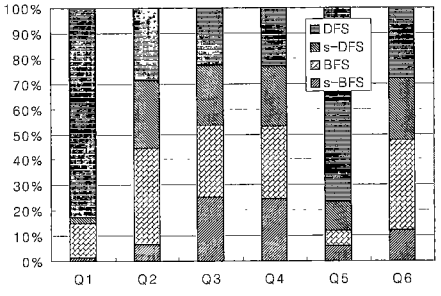
(c) 디스크 I/O 횟수



(d) 디스크 I/O 횟수 비율



(e) 디스크 동작 시간 (단위:초)



(f) 디스크 동작 시간 비율

그림 5 성능 비교

데이타베이스에서와 같이 각 노드가 하나의 객체로 저장되어 있을 경우에 시그니처를 이용하면 데이타베이스 내의 각 객체의 탐색 횟수를 줄일 수 있으므로 성능이 향상될 것이라는 것을 알 수 있다.

그림 5 (c), (d)는 데이타를 BFS, DFS로 클러스트링하여 저장하였을 경우 디스크 I/O의 횟수를 측정한 것이다. 이것을 보면 일반적인 경우에 BFS가 더 나은 성능을 보이는 것을 알 수 있다. 이것은 정규 경로식을 처리할 때 만족하지 않는 노드는 탐색할 필요가 없는 경우가 많이 발생하기 때문에 그래프 탐색이 항상 자식 방향으로 일어나지 않기 때문이다. 이때 시그니처를 이용하여 저장하였을 경우에 디스크 I/O가 현저히 적다는 것을 알 수 있다. 하지만 질의 결과가 클수록 디스크 I/O 성능은 줄어든다. (d)에서 Q3, Q4를 보면 시그니처를 이용한 것이 크게 나은 성능을 보이지 않는데 이것은 비록 노드의 탐색 횟수는 줄지만 각 노드에 시그니처를 저장함으로서 한 페이지에 저장할 수 있는 노드의 개수가 줄기 때문에 일정한 횟수 이상으로 노드 탐색이 많아지면 디스크 I/O가 더 많이 발생하기 때문이다. 하지만 이런 경우는 모든 데이타가 완전히 클러스트링이 되어 있다는 가정하에 수행한 결과이기 때문에 그 차이가 작은 것이며 실제 환경에서는 완벽한 클러스트링을 할 수 없으므로 시그니처를 한 경우와 사용하지 않은 경우의 차이는 더욱 커질 것으로 예상된다.

또한 시그니처를 이용하여 데이타를 저장할 경우 클러스트링은 BFS로 하는 것이 좋다는 것을 알 수 있다. 그러므로 한 노드를 삽입할 때 그 노드의 부모나 자식 노드가 저장된 페이지에 삽입하는 것보다는 형제 노드가 저장된 페이지에 삽입하는 것이 디스크 I/O를 줄일 수 있다는 것을 알 수 있다.

그림 5 (e), (f)는 디스크 연산에 소요되는 시간을 계산한 것으로서 디스크 I/O 횟수를 측정한 것과 유사한 형태의 그

래프를 얻을 수 있다. 결국 클러스트링 방법에 관계없이 디스크 I/O 횟수를 줄이는 것이 성능을 향상하는 방법이라는 것을 알 수 있으며 클러스트링은 DFS 보다 BFS 방법으로 하는 것이 더 나은 성능을 얻을 수 있다는 것을 알 수 있다.

7. 결론 및 향후 연구 방향

본 논문에서는 시그니처 기법을 이용한 XML 데이타의 저장 및 이에 대한 질의인 정규 경로식을 처리하는 기법에 대하여 설명하였다. 정규 경로식을 s-NFA를 이용하여 수행함으로써 그래프의 노드 탐색을 줄일 수 있으며 실제 디스크 접근 횟수도 많이 줄일 수 있었다. 이와 같은 기법은 인덱스를 사용하지 않고 그래프를 탐색하는 경우에 사용하는 기법이다. 반구조적 데이타에 대한 인덱스도 또한 그래프 형태의 데이타이므로 이러한 기법을 인덱스 구축에도 적용하여 사용할 수 있다. 이 경우 인덱스 노드의 탐색 범위를 줄일 수 있다. 또한 BFS로 클러스트링 하는 것이 DFS 보다 디스크 접근 횟수를 줄일 수 있는 것을 알 수 있었다. 그러므로 시그니처 기반의 그래프 탐색의 경우에는 부모 자식간의 노드 클러스트링 보다는 형제 노드간의 클러스트링이 더 큰 성능 향상을 보인다는 것을 알 수 있었다. 이것은 그래프 탐색 시에 특정 노드의 하위 트리를 탐색하지 않는 경우가 빈번히 발생하기 때문이다.

하지만 엘리먼트의 개수가 많아지거나 그래프의 깊이가 깊어질수록 상위 노드의 시그니처는 1로 세팅되는 비트 수가 많아져서 시그니처가 포화(saturation)될 가능성이 높아진다. 이것은 시그니처의 크기를 늘리거나 자식 노드들에 관한 시그니처의 개수를 논문에서 제시한 1개에서 여러 개로 늘리는 시그니처 분할(signature chopping)을 통해서 해결될 수 있다. 또한 DTD 정보와 시그니처를 동시에 이용하면 그래프 탐색 범위를 훨씬 많이 줄일 수 있을 것으로 생각된다.

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 and Jennifer Widom. Query Optimization for XML. *VLDB*, 1999.

[4] 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.

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

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

[7] 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.

[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] Walter W. Chang and Hans J. Schek. A Signature Access Method for the Starburst Database System. *VLDB*, 1989.

[11] Chris Faloutsos. Signature files: Design and Performance Comparison of Some Signature Extraction Methods. *SIGMOD*, 1985.

[12] Hwan-Seung Yong, Sukho Lee, and Hyoung-Joo Kim. Applying Signatures for Forward Traversal Query Processing in Object-Oriented Databases. *ICDE*, 1994.

[13] Roy Goldman and Jennifer Widom. DataGuides: Enabling Query Formulation and Optimization in Semistructured Databases. *VLDB*, 1997.

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

[15] Yannis Papakonstantinou, Hector Garcia-Molina, and Jennifer Widom. Object Exchange Across Heterogeneous Information Sources. *ICDE*, 1995.

[16] Jae-Mok Jeong, Sangwon Park, Tae-Sun Chung, and Hyoung-Joo Kim. XWEET: XML DBMS for Web Environment. *The First Workshop on Computer Science and Engineering 2000*, Seoul, Korea, pages 16--17, June 2000, http://oopsla.snu.ac.kr/xweet/xweet-eng.ps.

[17] R. Sacks-Davis, A. Kent, and K. Ramamohanarao. Multikey Access Methods Based on Superimposed Coding Techniques. *TODS*, 12(4), 1984.

[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] GMD-IPSI. GMD-ISPI XQL Engine. *http://xml.darmstadt.gmd.de/xql*, 2000.

[23] W3C. Document Object Model (DOM). *http://www.w3.org/DOM*, 2 2000.

[24] Peter Linz. *An Introduction to Formal Languages and Automata*. Houghton Mifflin Company, 1990.

[25] Chris Ruemmler and John Wilkes. An Introduction to Disk Drive Modeling. *IEEE Computer*, 27(3), March 1994.

박 상 원

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

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.