GDIT를 기반으로 한 구조적 문서의 효율적 검색과 갱신을 위한 인덱스 설계

김 영 자† · 배 종 민††

요 약

SGML이나 XML언어를 사용하여 작성된 구조적 문서들에 대한 정보검색 시스템들은 문서의 부분검색을 지원한다. 문서의 구조에 바탕을 둔 질의를 효율적으로 처리하기 위해서는 색인에 관련된 메모리 오버헤드를 줄여야 하고, 질의에 대한 응답시간이 빨라야 하고, 문서 구조에 바탕을 둔 다양한 유형의 사용자 질의를 지원해야 하며, 문서 구조에 대한 변경이 발생했을 때 색인 구조에 대한 변경사항을 최소화하여야 한다. 본 논문에서는 전체문서인스턴스트리 구조를 제안하고, 이를 기반으로 텍스트 레벨 엘리먼트만을 색인하여, 색인과 검색의 효율성을 유지하면서 자료의 추가나 삭제등의 갱신이 발생할 때, 갱신의 파장을 최소화시킬 수 있는 색인구조와 질의처리 알고리즘을 제시하고 그 성능을 분석한다.

An Indexing Scheme for Efficient Retrieval and Update of Structured Documents Based on GDIT

Young-Ja Kim† · Jong-Min Bae††

ABSTRACT

Information retrieval systems for structured documents which are written in SGML or XML support partial retrieval of document. In order to efficiently process queries based on document structures, low memory overhead for indexing, quick response time for queries, supports to powerful types of user queries, and minimal updates of index structure for document updates are required. This paper suggests the Global Document Instance Tree(GDIT) and proposes an effective indexing scheme and query processing algorithms based on the GDIT. The indexing scheme keeps up indexing and retrieval efficiency and also guarantees minimal updates of the index structure when document structures are updated.

1. 서 론

일반적으로 문서는 묵시적이든 명시적이든 다양한 구조 정보를 가지고 있다. 명시적으로 텍스트에 문서의 구조를 넣은 문서를 구조적 문서(structured document)라고 한다. SGML(Standard Generalized Markup Language)이나 XML(eXtensible Markup Language) 등의 마크업 언어들은 문서가 가지는 계층적 구조를 기술하여 엘리먼트들 사이의 구조적 연관성을 표현하고 이러한 구조정보는 텍스트를 브라우징이나 검색에 사용할 때 문서전체가 아닌 부분항목들로 다룸으로서 질의에서 찾고자 하는 특정 영역에 바로 접근할 수 있어 질의 발생시 사용자가 질의에서 원하는 특정 영역에 바로 접근할 수 있다.

기존의 데이타베이스 시스템이나 현재까지의 관습적

† 정 회 원 : 경상대학교 대학원 전자계산학과
†† 종신회원 : 경상대학교 컴퓨터과학과 교수/정보통신연구센터
논문접수 : 1999년 11월 6일, 심사완료 : 1999년 12월 24일

인 정보검색시스템들에서는 문서를 연속된 단어들의 집합으로 인식하며, 검색시에 문서를 검색단위로 간주하고 있어 문서에 내재되어 있거나 명시되어 있는 구조적 정보들은 시스템의 검색과정에서는 사용하지 않고 있다. 최근에 많은 문서들이 SGML이나 XML과 같은 마크업 언어를 사용하여 구조적문서로 생성됨에 따라 문서들의 임의 부분을 쉽게 검색하고 접근할 수 있는 구조기반 정보검색시스템들에 대한 연구가 진행되고 있다[3, 5, 6, 8, 12].

문서의 구조에 바탕을 두어서 문서의 일부를 검색하기 위해서는 문서의 구조에 대한 색인과 그 색인 구조에 바탕을 둔 질의 처리 알고리즘을 개발해야 한다. 문서 구조에 대한 색인을 할 때 주요 고려 사항은 첫째, 텍스트 속의 단어뿐 아니라 텍스트를 구성하고 있는 엘리먼트에 대하여 색인이 필요하기 때문에 색인에 관련된 메모리 오버헤드를 줄여야 하고, 둘째, 질의에 대한 응답시간이 빨라야 하고, 셋째, 문서 구조에 바탕을 둔 다양한 유형의 사용자 질의를 지원해야 하며, 넷째, 문서 구조에 대한 변경이 발생했을 때 색인 구조에 대한 변경사항을 최소화하여야 한다. 질의에 대한 응답시간을 빠르게 하기 위해서는 문서의 구조에 대하여 가능한 많은 정보를 색인 해야 하므로 색인을 위한 메모리 오버헤드가 늘어난다. 반면에 색인에 필요한 메모리 양을 줄이기 위해서 문서구조의 일부분만을 색인할 경우에는, 색인된 엘리먼트의 ID를 이용해서 찾고자 하는 엘리먼트의 ID를 계산해야 하므로 검색비용이 커진다.

본 논문에서는 색인 오버헤드를 줄이면서 자료의 추가나 삭제등의 갱신이 발생시 갱신의 파장을 줄이고 빠른 질의 응답시간을 보장하는 구조적 문서들에 대한 색인구조와 질의처리 알고리즘을 제시한다. 제안된 메카니즘의 기본적인 개념은 다음과 같다. 하나의 DTD (Document Type Definition)를 문서 구조로 가지는 모든 문서 인스턴스들에 대한 전체문서인스턴스트리(GDIT : Global Document Instance Tree)를 생성한다. GDIT란 문서 인스턴스의 구조를 트리로 표현했을 때, 모든 문서 인스턴스 트리의 합집합을 말한다. 그리고 문서 인스턴스에서 색인어를 포함하고 있는 텍스트 레벨 엘리먼트에 대해서만 색인을 한다. 질의 처리시에는 이미 색인된 텍스트 레벨 엘리먼트를 얻어서 이 텍스트 레벨 엘리먼트와 GDIT를 이용해서 찾고자 하는 문서상의 엘리먼트를 결정하여 대응되는 방에 그 단어의 빈도수를 누적시킨다. 이 방법은 모든 엘리먼트에 색인하는 구조기반 검색방법에 비하여 색인에 필요한 메모리와 시간 비용이 적다. 또한 기존의 텍스트 레벨 엘리먼트만 색인하는 방식에 비하여 색인에 필요한 메모리와 시간비용이 뒤떨어지지 않으면서 질의에서 찾는 엘리먼트의 레벨과 관계없이 질의처리시간이 빠르면서 일정하다. 특히 제안된 알고리즘은 문서의 갱신이 발생할 때, 갱신될 엘리먼트와 형제관계에 있는 엘리먼트들 중에서, 갱신될 엘리먼트 뒤에 위치한 동일한 이름을 가지는 엘리먼트들과 그 엘리먼트의 서브 트리에 대해서만 갱신이 발생하게 함으로서, 엘리먼트 갱신에 따른 갱신의 파장을 최소화시킬 수 있다.

본 논문의 구성은 다음과 같다. 2장에서는 관련연구에 대하여 논하고 3장에서는 질의를 분석하며 4장에서는 GDIT 구성방법과 그 의미에 대해서 논한다. 5장에서는 색인알고리즘을 제안한다. 6장에서는 검색과 갱신 알고리즘을 제시한다. 7장에서는 본 논문에서 제시된 색인 구조의 성능을 분석하고, 8장에서는 결론을 보인다.

2. 관련 연구

구조적 문서를 처리하기 위하여 문서구조 DTD를 데이타베이스 스키마로 매핑하고 이를 기반으로 구조기반질의를 지원하는 연구가 많이 진행되어 왔다[4, 5, 7, 8, 9, 10, 11]. 구조적 문서의 엘리먼트를 관계형 데이터베이스의 테이블로 매핑하는 시스템[4, 7, 10, 11]의 경우, 문서구조를 표현하는데 필요한 테이블과 튜플수가 많이 필요하며, 메모리 오버헤드를 줄이는 여러 가지 알고리즘들은 DTD가 가지고 있는 정보를 모두 표현하지 못한다. 문서구조 DTD를 객체지향 데이타베이스의 클래스로 매핑하는 시스템[5, 8, 9]의 경우, 구조적 문서를 OODBMS의 클래스들로 모델링하여, 문서에 나오는 단위만을 색인하여 구조적 문서 검색을 지원하는 시스템[8, 9]과 SGML문서를 O_2 OODB(Object-Oriented DataBase) 스키마로의 매핑하고, OODB질의어(O_2SQL, IQL calculus)를 확장한 시스템[5]이 있다. DTD를 다양한 모델의 스키마로 매핑하는 이러한 방법들은 매핑 과정에서 DTD가 가지고 있는 제약조건등을 무시하거나, 혹은 표현하지 못하는 경우가 있으며, 구조적 문서를 처리하기 위해서 시스템의 확장이 불가피하다[5, 7, 10].

구조 기반 검색의 기반이 되는 색인 구조로서, SGML 문서를 트리 형태로 표현하여, 문서에서 발생하는 모

든 엘리먼트를 대상으로 색인할 수 있다[1, 2]. 이 방법은 추출된 색인어가 발생된 엘리먼트의 모든 상위 엘리먼트에 대해서도 색인을 하게 되어 공간상의 중복이 발생하여 색인 오버헤드가 크다. 공간상의 중복을 줄이기 위하여 문서에서 발생하는 모든 엘리먼트의 색인을 피하는 방법으로서, 문서를 k-ary 완전트리로 표현하여, 임의의 엘리먼트의 하위엘리먼트 모두에 특정 색인어가 공통적으로 존재할 경우 그 색인어를 임의의 엘리먼트에만 기억시켜 색인하는 방법[3]과, 문서구조에서 색인어를 포함하고 있는 엘리먼트 중에서 색인어와 가장 가까운 레벨에 있는 엘리먼트만을 색인하는 방법[6]이 있다. 이 방법들은 자식 엘리먼트의 최대 수가 고정되어 있기 때문에, 엘리먼트 추가 시에 추가할 엘리먼트 이후의 모든 형제 엘리먼트에 대하여 그 위치 정보를 변경해야 하고, 자식 엘리먼트 수가 미리 정해진 한계를 넘을 경우, 모든 문서 인스턴스에서 발생되는 모든 엘리먼트들의 위치 정보를 변경해야 한다.

한편 문서를 내용 인덱스, 지역 구조 인덱스, 전체 구조 인덱스, 구조 메타 인덱스의 4가지 인덱스로 구성하고, 문서에 포함되어 있는 색인어와 엘리먼트들의 발생 위치를 ID로 사용함으로서, 엘리먼트나 색인어에 대한 갱신이 발생할 때, 갱신이 발생한 엘리먼트의 조상 엘리먼트들의 정보만을 변경하게 함으로서 구조적 문서들을 효율적으로 갱신하기 위한 색인 구조가 있다[12]. 이 방법은 내용 인덱스 구성시에 해당 색인어가 발생하는 위치만을 기록하고 색인어가 존재하는 엘리먼트들에 대한 정보를 넣지 않기 때문에 내용과 구조의 혼합 질의를 효과적으로 처리할 수 없고, 엘리먼트의 추가등의 문서 구조 갱신이 발생할 경우 갱신 엘리먼트 이후의 모든 엘리먼트들의 ID값을 변경해야 하며, 각 문서 인스턴스별로 문서 구조 정보를 가지기 때문에 메모리 오버헤드가 크다.

본 논문에서는 각 DTD기반의 문서 인스턴스의 집합에 대하여 하나의 GDIT를 구성하고, 문서 인스턴스에서 색인어를 포함하고 있는 텍스트 레벨 엘리먼트만을 색인하여, 색인 오버헤드를 줄이고 검색효율이 떨어지지 않으면서 문서의 갱신에 효율적인 인덱스 구조를 제시한다.

3. Global Document Instance Tree(GDIT)

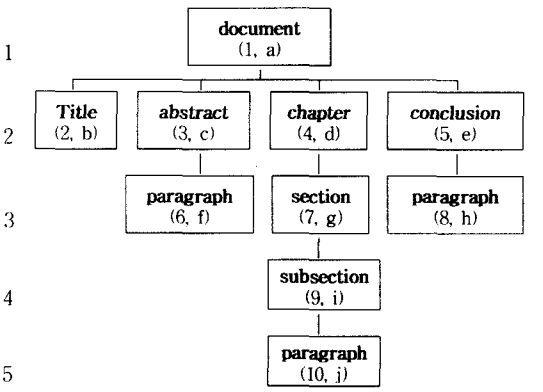
제안된 색인 구조는 GDIT에 기반을 두고 있다. 여기서는 GDIT의 구성 방법과 그것이 내포하고 있는 의미에 대하여 논한다.

3.1 GDIT 구성

GDIT는 문서 인스턴스의 구조를 트리로 표현했을 때 모든 문서 인스턴스 트리의 합집합을 말한다. GDIT를 구성하기 위해서 DTD 트리가 필요하다.

3.1.1 DTD 트리

(그림 1)과 같은 DTD 트리가 있다고 가정하자.



(그림 1) DTD 트리의 예

(그림 1)의 각 엘리먼트에는 순서쌍 (i, j)가 부여되어 있다. 여기서 i는 트리를 너비우선순회규칙에 따라서 차례로 부여된 번호이다. 이 번호를 앞으로 DEN(Dtd Element Number)라 칭한다. DEN은 나중에 루트에서부터 각 엘리먼트까지의 경로 정보를 나타내는 것으로 활용될 것이다. j는 문서 인스턴스에서 대응되는 엘리먼트가 실제로 발생된 횟수를 말한다. 각 엘리먼트별로 문서 인스턴스에서 발생하는 조상의 경로가 다르지만, IEN은 1번부터 차례대로 할당된다.

3.1.2 GDIT 구성 알고리즘

<알고리즘 1>은 GDIT를 구성하는 알고리즘을 개략적으로 보인 것이다.

<알고리즘 1> GDIT 구성 알고리즘

```
GDIT(targetElement, N, ANCESTORS){
(1)   if (The Nth targetElement having ANCESTORS as its
          ancestor does not exist in GDIT){
```

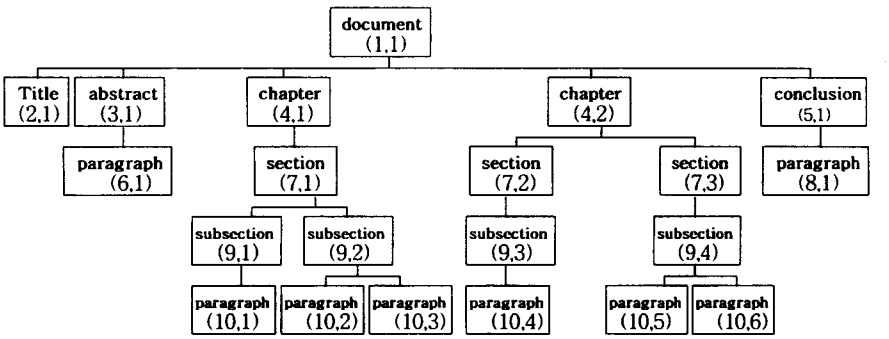
```
(2)     get the DEN of targetElement from the DTD tree;
(3)     increment j of (i, j) attached to the DTD node;
(4)     assign j to IEN of targetElement;
(5)     insert a targetElement with (DEN, IEN) into GDIT;
      }
    else
      return;
  }
```

GDIT를 구성하기 위해서는 모든 문서 인스턴스 트리를 합치고, 합쳐진 각 노드에 ID로서 사용될 (DEN, IEN) 값을 결정하는 두가지 작업이 필요하다. 여기서 DEN이란 3.1.1에서 제시한 DTD 엘리먼트 발생 순서를 말하고, IEN(Instance Element Number)이란 문서 인스턴스에서 해당 엘리먼트가 발생된 순서를 의미한다. 임의의 첫 번째 문서 인스턴스의 각 엘리먼트에 대하여, (그림 1)의 DTD트리로부터 DEN값을 알 수 있고, DTD 노드에 부여된 순서쌍의 두 번째 값 즉, 엘리먼트가 실제로 발생된 횟수를 1증가시키고 그 값을 IEN으로 부여한다. 모든 문서 인스턴스들을 대상으로 위와 같은 과정을 반복하면, 문서 인스턴스 구조의 루트에서 텍스트 레벨 엘리먼트까지의 각 경로별로 발생하는 가능한 경로 정보들을 모두 포함하는 트리인 GDIT가 구성된다.

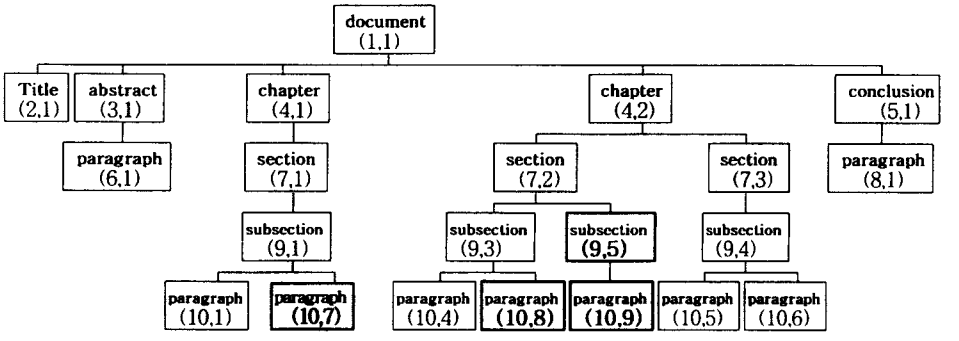
3.1.3 GDIT구성 예

(그림 1)의 DTD를 문서 구조로 가지는 두 개의 문서 인스턴스가 (그림 2)와 (그림 3)과 같이 있다고 가정하자.

문서 인스턴스 Doc1이 제일 처음 읽는 문서이므로, GDIT에는 아무것도 존재하지 않으며 (그림 1)의 DTD 트리의 각 엘리먼트가 가지는 정보 중 두 번째 항목 값은 모두 0이다. 문서 인스턴스 Doc1의 루트에서부터 문서를 읽으면서 엘리먼트가 발견되면 <알고리즘 1>의 GDIT구성 알고리즘을 사용하여 GDIT에 엘리먼트를 만든다. 예를 들어, 문서 인스턴스 Doc1의 루트 엘리먼트는 GDIT에 존재하지 않으므로(1), (그림 1)의 DTD트리에서 DEN번호 1을 가져오고(2), 1을 IEN값으로 한(4) (1, 1)을 가진 document엘리먼트를 GDIT에 넣는다(5). 나머지 남아 있는 모든 엘리먼트에 대해서 같은 과



(그림 2) 첫 번째 문서 인스턴스 구조 Doc1



(그림 3) 두 번째 문서 인스턴스 구조 Doc2

정을 수행하면, (그림 2)의 문서 인스턴스 구조 Doc1과 동일한 GDIT가 생성된다. (그림 2)에서 각 엘리먼트에는 순서쌍 (DEN, IEN)이 부여되어 있다. IEN에 대해서 보면 (그림 2)에서 두 개의 chapter 엘리먼트가 발생했으므로, 첫 번째 chapter 엘리먼트의 IEN값은 1이고 (3, 4), 두 번째 chapter 엘리먼트의 IEN값은 2로서(3, 4), IEN은 엘리먼트가 발생되는 순서를 나타내며, 엘리먼트 종류에 따라서 1번부터 차례대로 번호가 부여된다. 문서 인스턴스에서 발생하는 조상의 경로가 다르지만, IEN은 1번부터 차례대로 할당된다.

다음으로, 하나의 DTD에 속한 임의의 두 번째 문서 Doc2가 (그림 3)과 같이 있다고 가정하자.

문서 인스턴스 Doc2의 루트에서부터 문서를 읽으면서 엘리먼트가 발견되면 그 엘리먼트를 GDIT와 비교하여, 엘리먼트가 GDIT에 없을 경우 GDIT에 엘리먼트를 만든다. 루트엘리먼트는 GDIT에 존재하므로 새로 생성되지 않고 GDIT의 (DEN, IEN)값 (1, 1)을 그대로 사용한다. 레벨 5의 두번째 paragraph은 GDIT에 존재하지 않으므로(1), (그림 1)의 DTD트리에서 5레벨의 paragraph엘리먼트의 DEN번호 10과(2), 엘리먼트의 발생횟수가 6이므로 1증가시킨 값 7을 IEN값으로 가지는(4) paragraph엘리먼트를 GDIT에 생성한다. 문서 인스턴스 Doc2를 GDIT와 비교하여 같은 위치에 있지 않은 엘리먼트는 GDIT에 엘리먼트를 생성하고 새로운 (DEN, IEN)을 부여한다. 문서 인스턴스 Doc2를 모두 읽은 후의 GDIT는 (그림 4)와 같다.

계속해서, 새로 생성된 트리 (그림 4)와 남아있는 모든 문서 인스턴스를 대상으로 위와 같은 과정을 반복한다. 그 결과, 문서 인스턴스 구조의 루트에서 텍스트 레벨 엘리먼트까지의 각 경로별로 발생하는 가능한 경로 정보들을 모두 포함하는 트리가 생성된다. 이 트리가 GDIT이다.

3.2 GDIT의 노드에 대한 의미 부여

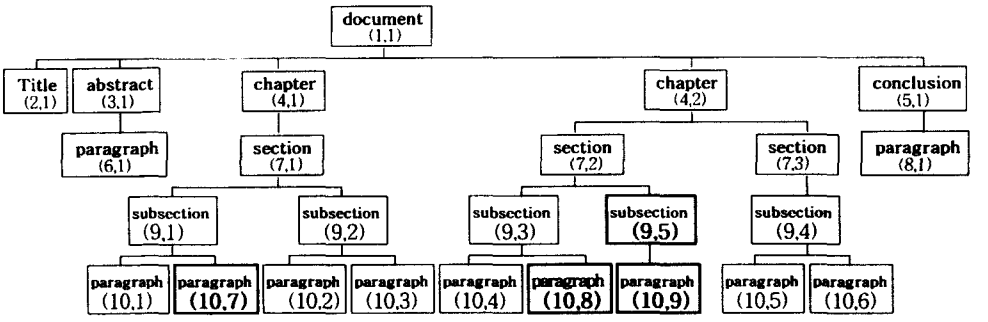
앞에서 논한 바와 같이, 하나의 DTD를 문서 구조로 가지는 두 개의 문서 인스턴스 구조를 조합하여 구조상 중복되는 엘리먼트들은 하나로 인식하여, 두 문서 구조의 합집합인 새로운 문서 구조를 만든다. 만들어진 문서 구조와 다음 문서 구조를 조합하여 구조상 중복되는 엘리먼트들은 하나로 인식하여 새로운 문서구조를 만든다. 이 과정을 모든 문서 인스턴스 구조에 반복한 결과로 생성된 트리를 GDIT라고 한다. 결국 GDIT는 모든 문서 인스턴스에서 발생된 엘리먼트의 갯수가 모두 반영되었다. 따라서 GDIT에서 부여된 값의 쌍 (*DEN, IEN*)은 문서내에서 엘리먼트의 위치를 나타내게 된다.

3.2.1 DEN(Document Element Number)의 의미

DTD트리의 구조는 인스턴스에 무관하게 변하지 않는 구조이기 때문에, 각 엘리먼트에 할당된 DEN에 일정한 의미를 부여할 수 있다. 여기서는 각 엘리먼트에 부여된 DEN값은, 그 엘리먼트의 트리상의 조상의 경로정보라는 의미를 부여한다. 예를 들어 (그림 1)에서, DEN 6은 조상의 경로가 document/abstract인 paragraph임을 의미하고, DEN 10은 조상의 경로가 document/chapter/section/subsection인 paragraph을 의미한다. 이 경로정보는 비록 DTD트리에서 유도된 정보이지만, 모든 문서 인스턴스에 대하여 변함없이 적용된다.

3.2.2 IEN(Instance Element Number)의 의미

DEN에는 발생지시자에 대한 정보가 포함되어 있지 않다. 따라서 실제 문서 인스턴스에서, 예를 들어 다수



(그림 4) 두개의 문서 인스턴스 Doc1과 Doc2에 대한 GDIT

의 chapter가 있고, 각 chapter에서는 다수의 section이 있을때, 특정 section이 속한 chapter를 DEN만으로는 결정할 수 없다. 이 문제를 해결하는 요소가 IEN이다.

GDIT는 모든 문서 인스턴스에서 발생된 모든 엘리먼트의 발생횟수 및 순서가 반영된 구조이다. GDIT는 문서 인스턴스에 있는 각 엘리먼트에 대하여 DEN의 경우와 마찬가지로 문서 구조가 고정되어 있기 때문에, 문서 인스턴스에 있는 각 엘리먼트에 부여된 IEN에 조상의 경로에 대한 의미를 부여할 수 있다. 예를 들어, 문서 인스턴스 집합에 대한 GDIT가 (그림 4)라고 가정했을 때, (DEN, IEN)값이 (10, 7)인 경우, IEN값 7의 의미는 임의의 문서 인스턴스에서 조상의 경로가 1-1-1-2임을 의미하는 것으로 고정된다.

3.2.3 엘리먼트 식별자(EID : Element IDentifier)

엘리먼트 식별자는 문서 인스턴스에서 발생되는 각각의 엘리먼트를 유일하게 식별하는데 사용된다. 엘리먼트 식별자의 구성요소는 (DEN, IEN)쌍에 문서 인스턴스 번호(DIN : Document Instance Number)를 추가하여 <DIN, (DEN, IEN)>로 표현된다.

4. 색인 구조

4.1 색인 알고리즘

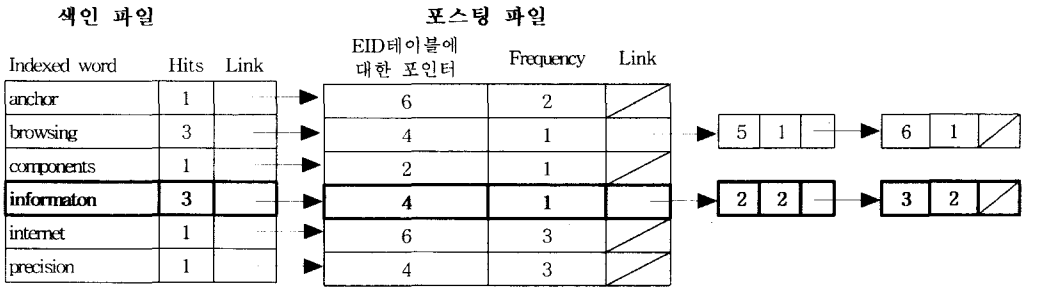
색인 알고리즘은 <알고리즘 2>와 같다. 문서에서 엘리먼트가 발견되면, <알고리즘 1>의 GDIT구성알고리즘을 사용하여 GDIT에 엘리먼트가 존재하지 않을 경우 엘리먼트를 생성하고 (DEN, IEN)을 부여한다. 엘리먼트가 텍스트 레벨 엘리먼트이면, 색인어와 엘리먼트 내의 색인어 출현빈도수를 엘리먼트의 EID와 함께 역리스트에 저장하고, 엘리먼트가 텍스트 레벨 엘리먼트가 아닐 경우에는 DEN과 DEN의 순서를 ANCESTORS에 저장하여 GDIT에서 엘리먼트를 찾을 때에 사용한다.

<알고리즘 2> 색인 알고리즘

```
index(DIN, targetElement, N, ANCESTORS){
  while(DIN is not indexed) {
    if (The Nth targetElement having ANCESTORS as its
        ancestor is text-level element) {
        get the (DEN, IEN) of targetElement from the GDIT;
        while((indexWord = read(DIN)) != newElement){
            if(indexWord is indexing word) {
              if(occurrence frequency of indexWord > 1)
                 increment of frequency of indexWord in
                  inverted list;
              else{
                 assign 1 to frequency of indexWord;
                 store <indexWord, <DIN, (DEN, IEN)>,
                  frequency> to inverted list;
              }
            }
        }
        targetElement = newElement;
        GDIT(targetElement, N, ANCESTORS);
    }
    Else{
        get the DEN of targetElement from DTD tree;
        delete (DEN, N)s except targetElement's ancestors
         in ANCESTORS;
        store (DEN, N) of targetElement to record array
         ANCESTORS;
        targetElement = read(DIN);
    }
  }
}
```

4.2 역리스트 구조

제안된 GDIT기반의 검색을 위한 색인 구조는 (그림 5)와 같다. 색인화일의 구성요소는 색인어, 색인어를 포함하고 있는 텍스트레벨 엘리먼트 갯수이고, 포스팅 파일의 구성요소는 색인어를 포함하고 있는 EID테이블에 대한 포인터, 색인어 빈도수로 구성된다. EID테이블



Hits : 색인어를 포함하고 있는 텍스트 레벨 엘리먼트 개수

(그림 5) 색인 구조

은 색인어를 포함하고 있는 EID를 저장하고 있는 테이블이다. 이때 유의할 사항은 문서 인스턴스 엘리먼트 중에서, 색인어를 포함하고 있는 텍스트 레벨 엘리먼트만 색인함으로서 메모리 오버헤드를 줄인다. 또한 포스팅 파일에서의 EID에 대한 중복 저장을 줄이고, 문서 구조의 수정이 발생할 때 한 텍스트 레벨 엘리먼트내에서 수정될 색인어의 개수와 무관하게 갱신을 처리하기 위하여 EID테이블을 따로 두었다.

번호	EID<DIN, (DEN, IEN)>
1	<1, (2, 1)>
2	**<1, (6, 1)>**
3	**<1, (8, 1)>**
4	**<1, (10, 1)>**
5	<1, (10, 4)>
6	<1, (10, 5)>

(그림 6) EID 테이블

5. 질의어 처리

5.1 검색

<Algorithm 3>은 앞에서 주어진 색인 구조를 바탕으로 사용자 질의어에 대한 검색 알고리즘의 일부이다. 제시된 알고리즘은 논의를 단순화시키기 위해서, 단일 키워드와 엘리먼트만으로 구성된 질의어에 대한 검색 알고리즘이다.

<알고리즘 3> 검색 알고리즘의 일부

```
retrieval(queryWord, queryElement){
(1)  get <keyword, frequency, <DIN, (DEN, IEN)> from the queryWord
     in the inverted list.
(2)  for each <keyword, frequency, <DIN, (DEN, IEN)>>{
(3)      get a ancestors' path of the (DEN, IEN) from the GDIT;
(4)      store the path into a record array PATH;
(5)      /* get the (DEN, IEN) of the queryElement */
(6)      while(the entry for the queryElement in the Element-DTD
           table is not empty){
(7)        get a tuple (DEN', Level) for the queryElement from the DTD
           tree;
(8)          If (DEN' == PATH[Level].DEN){
(9)            weight[PATH[Level]] += frequency;
(10)           break;
(11)         }
(12)     }
(13) }
(14)}
```

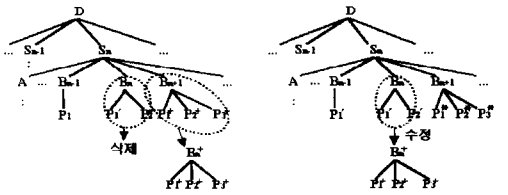
예를 들어, "information이라는 단어를 포함하고 있는 <section>을 검색하라."라는 질의어가 있다고 가정하자. (그림 5)의 역리스트에서 색인어 "information"은 3개의 문서 부분에 포함되어 있는데, 이를 <keyword, frequency, <DIN, (DEN, IEN)>>으로 표시하면 <information, 1, <1, (10, 1)>>, <information, 2, <1, (6, 1)>>, <information, 2, <1, (8, 1)>>이다(1). 우선 <information, 1, <1, (10, 1)>>에서 (10, 1)의 조상경로를 GDIT로부터 얻는다. (그림 4)의 경우 그 값은 (1, 1), (4, 1), (7, 1), (9, 1)이다(3). 이 경로값을 레코드형 배열 PATH에 저장한다(4). 찾고자 하는 엘리먼트는 <section>이므로, 경로 (1, 1), (4, 1), (7, 1), (9, 1)중에 찾고자 하는 <section>이 있는지를 알기 위하여 <section>의 DEN값을 구해야 한다(5). 그런데 한 DTD내에서도 <section>이 여러 곳에 발생할 수 있기 때문에 오직 배열 PATH에 포함되어 있는 <section>만을 찾아야 한다. 이를 위하여 DTD트리로부터 <section>에 대한 DEN과 <section>노드의 레벨을 구한다(7). 이들 각각에 대하여, 어느 것이 찾고자 하는 <section>인지 판단한다. 예를 들어 (그림 1)의 DTD트리에서 <section>의 (DEN', level)값은 (7, 3)뿐이다. PATH[3]의 값 (7, 1)의 DEN 7과 동일하므로(7), (7, 1)이 찾고자 하는 <section>의 <DEN, IEN>값이다. 따라서 (DEN, level)값이 (7, 1)인 <section>에 대응되는 가중치를 더한다(9). 단계(2)로 가서, 다음 원소에 대하여 이 과정을 반복한다.

5.2 삭제

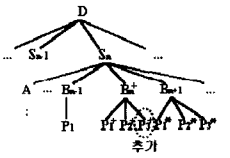
문서에서 엘리먼트를 삭제하는 과정은 크게 두 단계로 구분된다. 단계 1: 삭제될 엘리먼트의 하위에 있는 텍스트레벨 엘리먼트들의 EID <DIN, (DEN, IEN)>을 결정하고, 역리스트에서 색인어의 포스팅 엔트리 리스트를 조사하여 삭제할 엘리먼트의 EID와 같은 엔트리를 삭제한다. 단계 2: 엘리먼트의 삭제로 인하여 변경되는 엘리먼트의 순서정보를 수정한다. 예를 들어 첫 번째 chapter가 삭제되면, 두 번째 chapter가 첫 번째 chapter로 색인되어야 한다. 순서정보는 IEN과 연관되어 있다. 따라서 삭제할 엘리먼트의 뒤에서 발생하는 형제 엘리먼트중에서 이름이 같은 엘리먼트의 텍스트레벨 엘리먼트에 속한 색인어에 대응되는 EID테이블에서 IEN값을 수정한다. EID값을 수정할 때 필요하면 GDIT구조를 수정한다. 본 논문에서는 텍스트레벨 엘리먼트에 대해서만 색인을 하기 때문에, 텍스트레벨 엘리먼트가 아닌 것은 삭제대상에서 제외된다.

(그림 8)은 엘리먼트가 삭제되는 과정을 보인 것이

다. (a)에서 문서의 n번째 S엘리먼트인 S_n의 n번째 B엘리먼트 B_n을 삭제할 경우, B_{n+1}이 n번째 B엘리먼트 B_n이 된다. 새로운 B_n엘리먼트의 텍스트 레벨 엘리먼트에 (DEN, IEN)을 새로 부여하는 과정에서, (b)의 GDIT의 n번째 B엘리먼트 B_n은 2개의 P엘리먼트만 존재한다. GDIT 구성 알고리즘을 사용하여 새로운 P엘리먼트를 생성하고, (DEN, IEN)을 부여한다. 만일 부모엘리먼트가 B엘리먼트인 P엘리먼트의 개수가 k라면, 새로 생성되는 P엘리먼트의 IEN은 k+1이다. (c)는 수정된 GDIT이다. 역리스트에 나타나는 EID를 수정된 EID로 바꾸기 위해, EID테이블에서 엘리먼트의 EID를 수정한다. 이 경우 텍스트레벨 엘리먼트에 나타나는 색인어마다 포스팅 엔트리를 검색하여 EID를 수정하는 작업을 할 필요가 없다.



(a) 한 문서에서 삭제될 부분 (b) GDIT의 수정



(C) 수정된 GDIT

(그림 8) 엘리먼트 삭제과정

<알고리즘 4>는 문서구조를 삭제하는 알고리즘이다. 문서 DIN, 삭제하고자 하는 엘리먼트 deleteElement, root부터 deleteElement 상위 노드까지의 조상의 경로 ANCESTORS가 주어지면, 조상의 경로를 이용해서 GDIT로부터 삭제할 엘리먼트의 하위 문서 구조에서 텍스트 레벨 엘리먼트들을 찾고, 그 엘리먼트들의 (DEN, IEN)을 결정한다(2). 이 순서쌍과 같은 값을 가지는 색인어에 대하여 역리스트에서 해당 포스팅 엔트리를 삭제한다(8). 본 논문에서는 텍스트 레벨 엘리먼트에 대해서만 색인을 하기 때문에, 텍스트 레벨 엘리먼트가 아닌 것은 삭제 대상에서 제외된다(3). 다음으로 삭제할 엘리먼트의 뒤에서 발생하는 형제 엘리먼트중에서 이름이 같은 엘리먼트와 그 하위 문서구조에 존재하는 엘리먼트들은 EID가 변경된다. 이름이 같은 형제 엘리먼트의 텍스트레벨 엘리먼트에 대해서만 EID값을 수정한다(22).

<알고리즘 4> 삭제 알고리즘

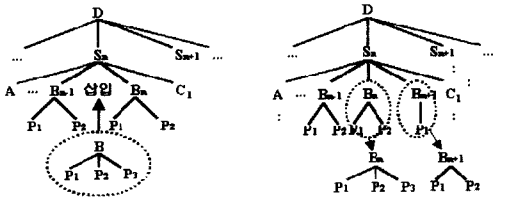
```
delete(DIN, deleteElement, N, ANCESTORS){
(1)   for (each text-level elements of deletetElement){
(2)       get the (DEN, IEN) of text-level Element from the GDIT;
(3)       for (each deleteWord of text-level Element){
(4)          /* delete an entry from inverted list */
(5)          while (1){
(6)           get a next entry <keyword, frequency, <DIN',(DEN',IEN')
                >> in inverted list matching with deleteWord;
(7)             if (<DIN, (DEN, IEN)> == <DIN', (DEN',IEN')>) {
(8)                delete the entry from the inverted list;
(9)                break;
(10)            }
(11)         }
(12)      }
(13)   }
(14)   for (each sibling targetElement of deleteElement of DIN){
       /* update IEN of the sibling element having same name with
       deleteElement. */
(15)      GDIT(targetElement, N, ANCESTORS);
(16)      get DEN of targetElement from the DTD tree;
(17)      if (DEN of deleteElement == DEN of targetElement) {
(18)         for (each sub elements of targetElement){
(19)            GDIT(targetElement, N, ANCESTORS);
(20)            if(sub element is text-level element){
(21)               get the (DEN, IEN) of the subElement from the
                     GDIT;
(22)               update EID of the subElement with <DIN, (DEN,
                     IEN)> in EID table;
                }
             }
          }
       }
    }
```

5.3 추가

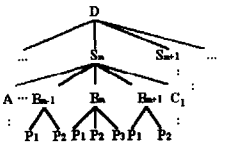
문서에 새로운 엘리먼트가 추가될 때에는 추가될 엘리먼트의 텍스트레벨 엘리먼트의 EID <DIN, (DEN, IEN)>을 결정하여 색인어를 역리스트에 추가하고, 필요시 GDIT 구조를 수정하고, 추가할 엘리먼트의 뒤에서 발생되는 형제엘리먼트중 이름이 같은 엘리먼트와 그 하위 문서 구조에 존재하는 엘리먼트에 대해서만 GDIT의 EID값을 수정한다.

(그림 9)는 엘리먼트가 추가되는 과정을 보인 것이다. (a)에서 문서의 n번째 S엘리먼트 S_n의 n-1번째 B엘리먼트 B_{n-1}와 n번째 B엘리먼트 B_n사이에 새로운 B엘리먼트를 추가할 경우, 추가되는 B엘리먼트는 n번째 B엘리먼트 B_n이 된다. 추가된 B엘리먼트의 하위 엘리먼트에 (DEN, IEN)을 부여하는 과정에서, (b)의 GDIT의 n번째 B엘리먼트는 2개의 P엘리먼트만 존재한다. GDIT 구성알고리즘을 사용하여 새로운 P엘리먼트를 생성하고 (DEN, IEN)을 부여한다. 만일 부모엘리먼트

가 B엘리먼트인 P엘리먼트의 개수가 k라면, 새로 생성되는 P엘리먼트의 IEN은 k+1이다. 세 번째 P엘리먼트의 EID를 구성되면, 색인어와 함께 역리스트에 추가한다. 다음으로 추가되는 엘리먼트 B_n뒤에 위치하는 형제엘리먼트 중 같은 이름을 가지는 B엘리먼트는 n+1번째 B엘리먼트 B_{n+1}이 된다. (b)의 GDIT의 n+1번째 B엘리먼트 B_{n+1}은 한 개의 P엘리먼트만 존재하므로, 새로운 엘리먼트를 생성하고 (DEN, IEN)을 부여하여, B엘리먼트의 두 번째 P엘리먼트의 EID를 구성한다. (c)는 수정된 GDIT이다. 역리스트에 나타나는 EID를 수정된 EID로 바꾸기 위해, EID테이블에서 엘리먼트의 EID를 수정한다. 이렇게 함으로서 텍스트레벨 엘리먼트에 나타나는 색인어마다 포스팅 엔트리를 검색하여 EID를 수정하는 작업을 할 필요가 없다.



(a) 한 문서에서 추가될 부분 (b) GDIT의 수정



(C) 수정된 GDIT

(그림 9) 엘리먼트 추가과정

<알고리즘 5>은 문서구조를 추가하는 알고리즘이다. 문서 DIN, 추가할 엘리먼트 insertElement, root부터 추가할 엘리먼트 상위노드까지의 조상의 경로 ANCESTORS가 주어지면, 먼저 <알고리즘 2>의 색인 알고리즘을 사용하여 추가할 엘리먼트의 텍스트 레벨 엘리먼트를 색인어와 함께 역리스트에 추가한다(1). 그 다음 insertElement뒤에 위치하는 엘리먼트 중에서 이름이 같은 엘리먼트들과 그 하위 문서구조에 존재하는 엘리먼트들은 문서에서 위치가 변경되는 엘리먼트들이므로 텍스트레벨 엘리먼트에 대해서만 EID가 수정되어야 한다. <알고리즘 1>의 GDIT 구성 알고리즘을 사용하여, GDIT에 존재하는 엘리먼트들은 IEN값만 변경되고, GDIT에 존재하지 않는 엘리먼트들은 GDIT에 엘리먼트를 추가하고, 새로운 (DEN, IEN)값을 부여한 다음, 그 값을 자신의 (DEN, IEN)으로 한 다음(7) (그림 6)의 EID테이블에서 엘리먼트의 <DEN, IEN>을 수정한다(10).

<알고리즘 5> 추가 알고리즘

```
   insert(DIN, insertElement, N, ANCESTORS){
(1) index(DIN, insertElement, N, ANCESTORS);
    /* update EID of the sibling element of insertElement having
     same name with insertElement */
(2) for (each sibling targetElement of insertElement of DIN){
(3)     GDIT(targetElement, N, ANCESTORS);
(4)     get DEN of targetElement from the DTD tree;
(5)     if (DEN of insertElement == DEN of targetElement)
(6)         for (each sub elements of targetElement){
(7)             GDIT(targetElement, N, ANCESTORS);
(8)             if(sub element is text-level element(
(9)                 get the (DEN, IEN) of the subElement from the
                    GDIT;
(10)                update EID of the subElement with <DIN, (DEN,
                    IEN)> in EID table;
                }
            }
        }
    }
}
```

6. 평가

다음은 본 논문에서 제시된 알고리즘들의 성능을 분석하기 위하여 사용되는 기호의 일부이다.

- h : 문서 인스턴스 트리의 높이
- k : 문서 인스턴스 트리의 차수
- m : 하나의 노드가 가지는 색인어의 평균 개수
- n_{doc} : 문서 인스턴스의 총 개수
- n_{key} : 색인어의 총 개수
- S_{EID} : EID(Element IDentifier)의 크기
- S_{key} : 색인어의 평균 크기
- S_{ptr} : 포인터 크기
- t_{seq} : 포스팅 파일에서 하나의 포스팅 엔트리에 접근하는데 걸리는 시간

6.1 색인비용

색인에 필요한 기억공간과 색인에 걸리는 시간으로 나누어서 분석한다.

6.1.1 기억공간

색인된 결과가 색인파일과 포스팅 파일을 사용하여 역리스트로 저장될 때, 색인에 필요한 공간 S_{index}은 색인파일에 필요한 공간 $S_{index-f}$ 와 포스팅 파일에 필요한 공간 $S_{posting-f}$ 의 합이다.

$$S_{index} = S_{index-f} + S_{posting-f}$$

색인파일에 필요한 공간 $S_{index-f}$는 색인어의 평균 크기와 포인터 크기를 더한 다음 색인어의 전체 개수를 곱한 것이다. 이는 모든 색인방법에 대하여 동일한 것으로 간주된다.

$$S_{index-f} = (S_{key} + S_{ptr}) * n_{key}$$

한편 포스팅 파일에 필요한 공간 $S_{posting-f}$는 색인어의 평균적인 포스팅 크기 S_{id}에 색인어의 전체개수를 곱한 것이다.

$$S_{posting-f} = n_{key} * S_{id}$$

여기서 색인어의 평균적인 포스팅 크기 S_{id}는 색인어의 평균적인 포스팅 개수 $N_{post-per-doc}$에 EID의 크기를 곱한 것이다.

$$S_{id} = N_{post-per-key} * S_{EID}$$

한 색인어에 대한 평균적인 포스팅 개수 $N_{post-per-key}$는 문서의 평균적인 포스팅 개수 $N_{post-per-doc}$에 문서 인스턴스의 총개수를 곱한 다음 색인어의 전체개수로 나눈 것이다.

$$N_{post-per-key} = [\frac{N_{post-per-doc} * n_{doc}}{n_{key}}]$$

단말노드의 색인어의 평균 개수가 m이라면, 레벨 $(h-1)$의 한 노드의 색인어의 갯수는 문서 인스턴스 트리의 차수에 단말노드의 색인어의 평균 개수를 곱한 $k*m$이고, 레벨 i의 임의의 노드의 색인어의 개수는 문서 인스턴스 트리의 차수의 $h-i$승에 단말노드의 색인어의 평균 개수를 곱한 $k^{h-i}*m$이다.

레벨 i의 노드의 개수는 k^{i-1}이므로, 레벨 i의 색인어의 갯수는 레벨 i의 임의의 노드가 가지는 색인어의 평균개수에 레벨 i의 노드 개수 k^{i-1}을 곱한 것이다.

$$k^{h-i} * m * k^{i-1}$$

$$= k^{h-1} * m$$

문서의 평균적인 포스팅 개수를 문서의 모든 레벨의 엘리먼트를 색인하는 방법과 본 논문에서의 GDIT를 사용하는 방법을 비교하면 다음과 같다.

(1) 문서의 모든 레벨의 엘리먼트를 색인하는 방법

문서의 평균적인 포스팅의 개수 $N_{post-per-doc}$는 문서의 각 레벨에서의 색인어의 개수를 모두 합한 것이다.

$$N_{post-per-doc} = \sum_{i=1}^{h} k^{h-1} * m$$

$$= h * k^{h-1} * m$$

이때 문서에서 색인어가 나타난 텍스트 레벨 엘리먼트의 모든 조상 엘리먼트들에 대해서도 색인하므로 중복색인이 된다. 그러므로 문서의 깊이가 깊거나 문서가 중첩구조를 가질수록 색인된 메모리는 커지게 되며 색인에 드는 메모리는 문서에 있는 엘리먼트의 갯수에 영향을 받는다.

(2) 텍스트 레벨 엘리먼트만을 색인하는 방법

본 논문에서의 문서의 평균적인 포스팅의 개수 $N_{post-per-doc}$는 레벨 i의 노드의 개수에 단말노드의 색인어 평균 개수를 곱한 것이다.

$$N_{post-per-doc} = k^{h-1} * m$$

색인어를 포함하는 텍스트 레벨 엘리먼트에 대해서만 색인하므로 색인에 드는 메모리는 텍스트 레벨 엘리먼트의 갯수에만 영향을 받는다. 텍스트 레벨 엘리먼트만 색인하는 방법은 문서의 모든 레벨의 엘리먼트를 색인하는 방법에 사용되는 메모리의 1/h정도의 메모리가 소요된다.

레벨우선순회규칙에 바탕을 두어서 텍스트 레벨 엘리먼트만을 색인하는 방법에서는 색인시에 ID이외에 문서구조에서의 엘리먼트 레벨, 문서 번호, 엘리먼트 타입 번호정보를 부가정보로 저장한다. 본 논문에서는 GDIT를 사용한 조상의 경로 정보를 나타내는 (DEN, IEN)을 ID로 사용하며, ID이외에 문서번호를 부가정보로 저장한다. 본 논문에서는 하나의 GDIT 트리가 필요하지만, 이 트리는 문서 인스턴스들의 엘리먼트 정보의 합집합이므로 많은 메모리를 필요로 하지 않는다.

텍스트 레벨 엘리먼트에 대해서만 색인하는 방법들은, 색인시에 텍스트 레벨 이외의 엘리먼트들은 색인하지 않는다. 그러므로 질의에서 찾는 엘리먼트를 처리하기 위해서는 BUS[6]에서처럼 자식 엘리먼트의 ID에서 부모 엘리먼트의 ID를 공식으로 구할 수 있거나, 아니면 각 문서 인스턴스마다 문서의 구조에 대한 정보를 가지고 있어야 조상에 대한 엘리먼트 정보를 알 수 있다. 그러나 본 논문에서는 GDIT를 사용하여 텍스트 레벨 엘리먼트에 대해서만 색인하며, 문서 인스턴스의 구조를 기억하기 위해 부가적인 메모리가 필요하지 않다.

6.1.2 색인시간

(1) 문서의 모든 레벨의 엘리먼트를 색인하는 방법

하나의 포스팅 엔트리를 색인하는 시간이 T_i라면, 문서의 모든 엘리먼트를 색인하는 방법에서는 색인어가 있는 텍스트 레벨 엘리먼트의 모든 조상 엘리먼트에 대해서도 색인하므로, 문서의 평균적인 색인시간 $T_{index-per-doc}$는 각 레벨의 포스팅의 개수를 합한 결과에 하나의 포스팅 엔트리를 색인하는 시간을 곱한 것이다.

$$T_{index-per-doc} = h * k^{h-1} * m * T_i$$

(2) 텍스트 레벨 엘리먼트만을 색인하는 방법

색인어가 있는 텍스트 레벨 엘리먼트에 대해서만 색인하므로, 문서의 평균적인 색인시간 $T_{index-per-doc}$는 임의의 레벨에서의 포스팅의 개수에 하나의 포스팅 엔트리를 색인하는 시간을 곱한 것이다.

$$T_{index-per-doc} = k^{h-1} * m * T_i$$

텍스트 레벨 엘리먼트만 색인하는 방법은 문서의 모든 레벨의 엘리먼트를 색인하는 방법보다 1/문서의 깊이(h)로 색인시간이 줄어든다. 색인에 관련된 공간과 시간에 대한 분석을 요약하면 <표 1>과 <표 2>와 같다. 아울러 엘리먼트 식별자의 표현에 대한 비교는 <표 3>과 같다.

〈표 1〉 색인에 필요한 공간과 시간의 비교(Ⅰ)

제안된 GDIT기반으로 텍스트 레벨 엘리먼트만을 색인하는 방법 : A
모든 엘리먼트를 색인하는 방법 : B

	A	B
문서마다 평균적인 포스팅의 갯수 ($N_{post-per-doc}$)	$k^{h-1} * m$	$h * k^{h-1} * m$
문서마다 평균적인 색인시간 ($T_{index-per-doc}$)	$k^{h-1} * m * T_i$	$h * k^{h-1} * m * T_i$

〈표 2〉 색인에 필요한 공간과 시간의 비교(Ⅱ)

제안된 GDIT기반으로 텍스트 레벨 엘리먼트만을 색인하는 방법 : A
레벨우선순회규칙에 바탕을 두어서 텍스트 레벨 엘리먼트만을 색인하는 방법 : C

	A	B
문서마다 평균적인 포스팅의 갯수 ($N_{post-per-doc}$)	$k^{h-1} * m$	$k^{h-1} * m$
문서마다 평균적인 색인시간 ($T_{index-per-doc}$)	$k^{h-1} * m * T_i$	$k^{h-1} * m * T_i$

〈표 3〉 엘리먼트 식별자 비교

제안된 GDIT기반으로 텍스트 레벨 엘리먼트만을 색인하는 방법 : A
레벨우선순회규칙에 바탕을 두어서 텍스트 레벨 엘리먼트만을 색인하는 방법 : C

	A	C
EID	(문서번호, ID)	(문서번호, ID, 엘리먼트레벨, 엘리먼트 타입)
ID	(DEN, IEN)	임의의 일련번호

6.2 검색 비용

특정 색인어가 존재하는 엘리먼트를 검색하는 질의처리에 필요한 검색시간 T_{search}은 색인파일 접근시간 $T_{index-f}$, 포스팅 리스트 접근시간 $T_{posting-f}$, 관련 포스팅 엔트리에서의 색인어의 빈도수를 찾고자 하는 엘리먼트에 해당하는 방에 누적시키는 시간 T_a의 합이다.

$$T_{search} = T_{index-f} + T_{posting-f} + T_a$$

색인어의 포스팅 리스트 접근시간 $T_{posting-f}$은 다음과 같다.

$$T_{posting-f} = (N_{post-per-key} - 1) * t_{seq}$$

색인어마다 포스팅의 개수 $N_{post-per-key}$는 다음과 같다.

$$N_{post-per-key} = [\frac{N_{post-per-doc} * n_{doc}}{n_{key}}]$$

(1) 문서의 모든 레벨의 엘리먼트를 색인하는 방법

문서내의 모든 엘리먼트들에 대해 색인하므로 관련 포스팅 엔트리에서의 색인어의 빈도수를 찾고자 하는 엘리먼트에 해당하는 방에 누적시키는 시간 T_a가 필요없으므로 엘리먼트를 검색하는 질의처리에 필요한 검색시간 T_{search}은 색인파일 접근시간 $T_{index-f}$과 포스팅 리스트 접근시간 $T_{posting-f}$의 합이다.

$$T_{search} = T_{index-f} + T_{posting-f}$$

(2) 레벨우선순회규칙에 바탕을 두어서 텍스트 레벨 엘리먼트만을 색인하는 방법

레벨 n의 임의의 찾고자 하는 엘리먼트의 ID를 구하기 위해서는 $\sum_{i=1}^{h-n} k^i$번의 부모 엘리먼트의 ID를 구하는 과정과 k^{h-n}번의 빈도수 누적과정을 수행해야 한다. 관련 포스팅 엔트리에서의 색인어의 빈도수를 찾고자 하는 엘리먼트에 해당하는 방에 누적시키는 시간 T_a는 부모 엘리먼트의 ID를 구하는데 걸리는 시간 T_{parent}에 $\sum_{i=1}^{h-n} k^i$를 곱한 것과 단말 노드에서의 색인어의 빈도수를 찾고자 하는 엘리먼트에 해당하는 방에 누적시키는 시간 T_{accum}에 k^{h-n}을 곱한 것을 더한 것이다.

$$T_a = \sum_{i=1}^{h-n} k^i * T_{parent} + k^{h-n} * T_{accum}$$

질의에서 찾고자 하는 엘리먼트 레벨과 색인어가 존재하는 텍스트 레벨과의 차이가 크면 클수록 부모 엘리먼트의 ID를 구하는 연산과정이 많아지므로 검색시간이 길어진다.

(3) 제안된 GDIT기반으로 텍스트 레벨 엘리먼트만을 색인하는 방법

본 논문에서는 모든 문서 인스턴스에 발생된 엘리먼트의 위치를 기억하고 있는 GDIT의 각 엘리먼트에 부여된 값의 쌍 (DEN, IEN)이 조상 엘리먼트들의 (DEN, IEN)을 나타내도록 하고 엘리먼트의 ID로 사용함으로서, 텍스트 레벨 엘리먼트에 부여된 (DEN, IEN)에서 조상 엘리먼트들의 (DEN, IEN)을 알 수 있다. 그러므로 텍스트 레벨 엘리먼트에서 질의에서 찾는 엘리먼트의 (DEN, IEN)을 바로 알 수 있다. 레벨 n의 임의의 찾고자 하는 엘리먼트의 ID를 구하기 위해서는 k^{h-n}번의 질의에서 찾는 엘리먼트의 ID를 구하는 과정과 누적연산 과정이 필요하다.

GDIT를 사용하여 관련 포스팅 엔트리에서 찾고자 하는 엘리먼트의 ID를 구하는데 걸리는 시간이 $T_{ancestor}$라면, 관련 포스팅 엔트리에서의 색인어의 빈도수를 찾고자 하는 엘리먼트에 해당하는 방에 누적시키는 시간 T_a는 $T_{ancestor}$에 k^{h-n}를 곱한 것과 단말 노드에서의 색인어의 빈도수를 찾고자 하는 엘리먼트에 해당

〈표 4〉 검색시간에 대한 비교(Ⅰ)

제안된 GDIT기반으로 텍스트 레벨 엘리먼트만을 색인하는 방법 : A
모든 엘리먼트를 색인하는 방법 : B

	A	B
검색시간 (T_{search})	$T_{index-f} + T_{posting-f} + T_a$	$T_{index-f} + T_{posting-f}$
포스팅 리스트 접근시간 ($T_{posting-f}$)	$[\frac{k^{h-1} * m * n_{doc}}{n_{key}} - 1] * t_{seq}$	$[\frac{h * k^{h-1} * m * n_{doc}}{n_{key}} - 1] * t_{seq}$

〈표 5〉 검색시간에 대한 비교(Ⅱ)

제안된 GDIT기반으로 텍스트 레벨 엘리먼트만을 색인하는 방법 : A
레벨우선순회규칙에 바탕을 두어서 텍스트 레벨 엘리먼트만을 색인하는 방법 : C

	A	C
검색시간 (T_{search})	$T_{index-f} + T_{posting-f} + T_a$	$T_{index-f} + T_{posting-f} + T_a$
포스팅 엔트리의 색인어의 빈도수를 해당하는 방에 누적시키는 시간 (T_a)	$k^{h-n} * T_{ancestor} + k^{h-n} * T_{accum}$	$\sum_{i=1}^{h-n} k^i * T_{parent} + k^{h-n} * T_{accum}$

하는 방에 누적시키는 시간 T_{accum}에 k^{h-n}을 곱한 것을 더한 것이다.

$$T_a = k^{h-n} * T_{ancestor} + k^{h-n} * T_{accum}$$

이 방법은 레벨우선순회 규칙으로 ID를 할당하는 방식과 같이 부모 레벨의 엘리먼트의 ID를 구하는 과정을 반복하지 않아도 되며, 질의에서 찾는 엘리먼트 레벨과 색인어를 포함하는 텍스트 레벨과의 차이에 영향을 받지 않으며 검색시간이 동일하게 소요된다.

6.3 갱신 비용

(1) 레벨우선순회규칙에 바탕을 두어서 텍스트 레벨 엘리먼트만을 색인하는 방법

레벨우선순회규칙에 바탕을 두어서 텍스트 레벨 엘리먼트만을 색인하는 경우에는, 레벨 n의 임의의 위치에 엘리먼트의 추가가 발생할 경우, 추가된 엘리먼트의 ID가 g라면, 갱신이 발생하는 엘리먼트는 추가된 엘리먼트 뒤에 위치하는 형제 엘리먼트들이며 그 개수 $N_{sibling-s}$는 다음과 같다.

$$N_{sibling-s} = k-((g-1)\ MOD\ k)$$

단말노드의 색인어의 개수가 m이라면, 각 형제 노드의 색인어의 개수는 다음과 같다.

$$k^{h-n} * m$$

갱신이 발생하는 포스팅의 개수는 갱신이 발생하는 형제 엘리먼트의 개수에 각 형제노드의 색인어의 개수를 곱한 것이다.

$$N_{updating-post} = (\ k^{h-n} * m) * N_{sibling-s}$$

위와 같이 레벨우선순회 규칙으로 ID를 할당하는 경우에는, 추가된 엘리먼트뒤에 위치하고 있는 모든 형제 엘리먼트의 ID가 변경되어야 한다. 문서 인스턴스에서 생길 수 있는 자식 엘리먼트의 최대 수를 고정시키므로, 엘리먼트의 추가가 인하여 문서에서 정한 자식 엘리먼트의 최대 수를 넘게 되는 경우에는, 갱신이 발생한 문서뿐만 아니라 저장되어 있는 모든 문서 인스턴스의 엘리먼트들을 갱신해야 하기 때문에 갱신해야 할 엘리먼트의 개수는 다음과 같다.

$$k^{h-1} * m * n_{doc}$$

(2) 제안된 GDIT기반으로 텍스트 레벨 엘리먼트만을 색인하는 방법

본 논문에서 제시된 방법에서는, 레벨 n의 임의의 위치에 엘리먼트 추가가 발생할 경우, 갱신이 발생되는 엘리먼트 개수는 다음과 같다.

$$0 \le N_{sibling-s} \ge k-((g-1)\ MOD\ k)$$

(여기서 g는 레벨우선순회규칙으로 엘리먼트에 번호를 할당했을 때 추가된 엘리먼트 번호이다.) 이는 갱신되어야 하는 대상이 추가된 엘리먼트와 같은 이름을 가지는 형제 엘리먼트 중에서, 추가될 엘리먼트의 뒤에 위치하는 엘리먼트뿐이기 때문이다.

단말노드의 색인어의 개수가 m이라면, 각 형제 노드의 색인어의 개수는 다음과 같다.

$$k^{h-n} * m$$

갱신이 발생하는 포스팅의 개수는 갱신이 발생하는 형제 엘리먼트의 개수에 각 형제노드의 색인어의 개수를 곱한 것이다.

$$N_{updating-post} = (\ k^{h-n} * m) * N_{sibling-s}$$

색인시에 색인어와 해당 텍스트 레벨 엘리먼트를 같이 색인하므로, 텍스트 레벨 엘리먼트의 ID를 갱신하려면 역리스트에서 텍스트 레벨 엘리먼트에 존재하는 색인어마다 갱신을 수행해야 한다. 본 논문에서는 (그림 5)의 EID테이블을 사용하여 역리스트에 텍스트 레벨 엘리먼트의 EID대신에 텍스트 레벨 엘리먼트의 EID에 대한 포인터를 둠으로서, 텍스트 레벨내에 있는 색인어의 개수 m에 영향을 받지 않고, 텍스트 레벨 엘리먼트 갯수만큼만 역리스트에서 갱신을 수행하면 된다. 즉 실제 갱신이 발생하는 포스팅의 개수는 갱신이 발생하는 형제 엘리먼트의 개수 $N_{sibling-s}$에 각 형제노드의 단말노드의 개수 k^{h-n}을 곱한 것이다.

$$N_{updating-post} = k^{h-n} * N_{sibling-s}$$

이상을 요약하면 <표 6>과 같다.

〈표 6〉 갱신 비용에 대한 비교

제안된 GDIT기반으로 텍스트 레벨 엘리먼트만을 색인하는 방법 : A
레벨우선순회규칙에 바탕을 두어서 텍스트 레벨 엘리먼트만을 색인하는 방법 : C

	갱신이 발생한 포스팅 엔트리의 갯수($N_{updating-post}$)	
	차수 K를 넘지 않을 경우	차수 K를 넘을 경우
A	$k^{h-n} * N_{sibling-s}$ $0 \leq N_{sibling-s} \geq k-((g-1) MOD k)$	$k^{h-n} * N_{sibling-s}$ $\leq N_{sibling-s} \geq k-((g-1) MOD k)$
C	$(k^{h-n} * m) * [k-((g-1) MOD k)]$	$k^{h-1} * m * n_{doc}$

7. 결 론

XML로 작성된 문서는 HTML과 달리 문서의 논리적 구조를 정의한다. 따라서 XML과 같은 구조적 문서의 특성을 활용한 정보검색 시스템을 개발하는 연구가 필요하다. 문서의 구조에 기반을 둔 검색시스템을 개발하기 위해서는 문서의 구조를 나타내는 엘리먼트들을 추가로 색인해야 하는데, 이에 따르는 메모리 오버헤드 및 검색비용이 증가된다. 본 논문에서는 GDIT를 제안하고, 이를 기반으로 한 색인 및 검색, 갱신 알고리즘을 제시하고 그 성능을 분석하였다. 제시된 알고리즘은 색인과 검색시간의 효율성을 유지하면서, 특히 문서구조의 갱신을 효율적으로 처리한다. 제시된 알고리즘은 문서의 내용과 구조가 혼합된 질의어를 처리한다. 앞으로 순수구조에 대한 질의처리 방법 및 문서구조의 효율적인 저장구조에 대한 연구가 더 필요하다.

참 고 문 헌

[1] 이희주, 장재우, 심부성, 주종철, "구조화 문서를 위한 정보 검색 인덱스의 설계", 정보과학회 가을 학술발표논문집 Vol.24, No.2, 1997.

[2] B. Lowe, J. Zobel and R. Sacks-Davis, "A Formal Model for Databases of Structured Text," In Proc. DASFAA'95, 1995.

[3] Y. K. Lee, S. J. Yoo, K. Yoon, and P. B. Berra, "Index Structure for Structured Documents," in Proc. Digital Libraries, 1996.

[4] I. A. Macleod, "Storage and retrieval of structured documents. Information Processing and Management," 26(2), 1990.

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

[6] D. W. Shin, H. C. Jang, H. L. Jin, "BUS : An Effective Indexing and Retrieval Scheme in Structured Documents," in Proc. Digitial Libraries, 1998.

[7] R. Sacks-Davis, A. Kent, K. Ramamohanarao, J. Thom, and J. Zobel, "Atlas : a nested relational database system for text applications," Technical Report CITRI/TR-92-52, Collaborative Information Technology Research Institute, Melbourne, Australia, 1992.

[8] M. Volz, K. Aberer and K. Bohm, "A Flexible Approach to Combine IR Semantics and Database Technology and its Application to Structured Document Handling," GMD Technical Report No. 891, 1995.

[9] M. Volz, K. Aberer and K. Bohm, "Applying a Flexible OODBMS-IRS-Coupling to Structured Document Handling," In Proceedings of 12th ICED, 1996.

[10] J. A. Thom, A. J. Kent, and R. Sacks-Davis, "TQL : A nested relational query language," Australian Computer Journal, 23(2), 1991.

[11] G. E. Blake, M. P. Consens, P. Kilpelainen, P. Larson, T. Snider, and F. Tompa, "Text/Relational database management systems : Harmonising SQL and SGML," In Proc. Int. Conf. on Applications of Databases, no. 819 in Lecture Notes in Computer Science, pages 267-280, 1994.

[12] K. Hirotaka, K. Hiroyuki, K. Hiroko and Y. Masatoshi, "An Efficiently Updatable Index Scheme for Structured Document," in proc. 9th International Workshop on Database and Expert Systems Application(DEXA), 1998.

김 영 자

e-mail : cybercpa@chollian.net

1991년 경상대학교 전산통계학과 졸업(이학사)

1993년 경상대학교 대학원 전자계산학과 졸업(공학석사)

1994년~현재 경상대학교 대학원 전자계산학과 박사과정

1993년~현재 동주여자상업고등학교 재직

-관심분야 : 병렬프로그래밍 언어, 디지털 도서관, 구조 정보검색

배 종 민

E-mail : jmbae@nongae.gsnu.ac.kr

1980년 서울대학교 사범대학 수학과 졸업(학사)

1983년 서울대학교 계산통계학과 이학석사(전산학)

1995년 서울대학교 계산통계학과 이학박사(전산학)

1982년~1984년 한국전자통신연구원 연구원

1984년~현재 경상대학교 컴퓨터과학과 교수/정보통신 연구센터

관심분야 : 병렬 프로그래밍 언어, 디지털 도서관, 정보검색