XML 문서의 분할 인덱스 기법

김종명[†] · 진 민[††]

요 약

기존의 *Numbering* 스킴을 이용한 XML 데이터의 인덱스 정의 방법은 개체가 삽입, 삭제, 갱신될 경우 인덱스가 재정의되어야 하는 문제점이 있다. 이러한 문제를 해결하기 위해서 본 논문은 블록단위의 인덱스 기법을 제안한다. 블록과 블록사이는 많아야 하나의 관계가 유지되도록 XML 데이터를 블록단위로 나누고, 각 블록에 대해 *Numbering* 스킴을 적용하여 인덱스를 정의한다. 이렇게 정의된 인덱스는 XML 데이터의 삽입, 삭제, 갱신의 경우 인덱스 재정의에 따른 오버헤드를 상당히 줄일 수 있다. 또한 XML질의를 지원하기 위해 정의된 인덱스를 이용하여 두 개체사이의 관계를 검색하는 *Parent-Child Block Merge Algorithm*과 *Ancestor-Descendent Block Merge Algorithm*을 제안한다. 그리고 *Ancestor-Descendent*관계를 빠르게 검색하기 위하여 블록식별자가 부모의 정보를 유지하는 방법을 소개하고 이를 이용한 *Parent-Child Block Merge Algorithm*과 *Ancestor-Descendent Block Merge Algorithm*을 제안한다.

Indexing Methods of Splitting XML Documents

Jong-Myung Kim[†] and Min Jin[††]

ABSTRACT

Existing indexing mechanisms of XML data using numbering scheme have a drawback of rebuilding the entire index structure when insertion, deletion, and update occurs on the data. We propose a new indexing mechanism based on split blocks to cope with this problem. The XML data are split into blocks, where there exists at most a relationship between two blocks, and numbering scheme is applied to each block. This mechanism reduces the overhead of rebuilding index structures when insertion, deletion, and update occurs on the data. We also propose two algorithms, *Parent-Child Block Merge Algorithm* and *Ancestor-Descendent Algorithm* which retrieve the relationship between two entities in the XML hierarchy using this indexing mechanism. We also propose a mechanism in which the identifier of a block has the information of its parents' block to expedite retrieval process of the ancestor-descendent relationship and also propose two algorithms, *Parent-Child Block Merge Algorithm* and *Ancestor-Descendent Algorithm* using this indexing mechanism.

Key words: XML 인덱스, Numbering 스킴, 블록분할, Ancestor-Descendent Block Merge

1. 서 론

XML이 갖는 데이터 표현능력과 교환의 용이성으로 XML의 사용이 지속적으로 증가하고 있어 XML 데이터 유지와 관리의 필요성이 요구되고 있다. 그러나 계층적인 구조를 갖는XML 문서를 정형화된 평면 구조를 갖는 기존의 관계데이터베이스에 저장하는 것은 쉬운 일이 아니다. XML 문서는 관계데이터베이스에 텍스트형태로 저장되거나 관계데이터베이스의 스키마 구조로 변환되어 저장된다. XML문서를 관계데이터베이스에 저장하는데 있어 구조정보를 유지하면서 관계데이터베이스의 스키마 구조로 변환하여 저장하는 것과 XML 질의 처리를 위한 개체

이 논문은 정보통신부의 정보통신 학술기초연구지원사업(정보통신연구진흥원)으로 수행한 연구결과입니다.
접수일 : 2002년 10월 24일, 완료일 : 2002년 12월 20일
[†] 정회원, 경남대학교 컴퓨터공학과 석사
[††] 종신회원, 경남대학교 정보통신공학부 교수

간의 관계를 빠르게 검색하는 것이 중요한 과제이다[3,4,7,9].

XML 질의는 일반적으로 XML 데이터의 계층적인 구조에 기반을 두고 있는데 계층적인 구조는 크게 *Parent-Child*와 *Ancestor-Descendant*의 두 가지 관계로 표현된다. 표 1의 XML 문서를 트리구조로 나타낸 그림 1에서 <*book*>엘리먼트는 서로 다른 여러 엘리먼트를 포함하고 있으며 <*Allauthors*> 엘리먼트는 2개의 <*author*> 엘리먼트를 포함하고 있다.

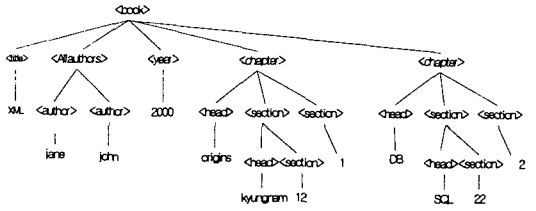


그림 1. 표 1의 트리 구조

XML 질의의 한 종류인 *Xpath*는 XML 데이터 검색을 위해서 *path*형식을 사용한다. 예를 들어 다음의 질의를 살펴보자[5,10].

book / chapter // head

이 질의는 *book* 엘리먼트의 *Child*인 *chapter*와 *Descendant*의 관계인 *head*를 검색한다. 다시 말해서 *book/chapter*와 *chapter//head*을 검색하고 이것을 다시 조인하는 것과 같다고 할 수 있다. *Xpath*로 표현된 XML 질의에서 알 수 있듯이 개체와 개체사이의 계층적인 구조를 빠르게 검색하는 것이 XML 질의 처리에 있어 주요관건이 된다.

본 논문에서는 XML 데이터를 관계데이터베이스의 스키마 구조로 변환하여 저장하고 검색하는데 필요한 블록분할 인덱스 기법과 정의된 인덱스를 이용하여 XML 데이터의 계층 구조를 쉽게 검색할 수 있는 합병조인알고리즘을 제안한다.

본 논문의 구성은 다음과 같다. 2장에서는 기존의 인덱스 기법과 인덱스를 이용한 XML 질의 처리 과정을 알아보고 문제점을 제시한다. 3장에서는 기존의 XML 인덱스 기법의 문제점을 해결하기 위한 블록단위의 인덱스기법을 소개하며, 4장에서는 XML 질의 처리를 위한 *Parent-Child Block Merge Algorithm*과 *Ancestor-Descendant Block Merge Algorithm*을 제안하며, 5장에서는 결론과 향후연구과제에 대해 기술한다.

2. 관련연구

2.1 Numbering 스킴

XML 데이터의 유지와 관리, 저장의 필요성에 따라 여러 가지의 데이터 저장방법과 효과적인 XML 질의 처리방법이 제안되고 있다. 그 중 *Numbering* 스킴[1,3,5,7,9]를 이용한 XML 인덱스 기법은 XML 데이터의 계층적인 구조정보를 유지함으로써 XML 데이터의 저장과 질의 처리에 상당히 효과적이다. 깊이우선순회방법을 사용하는 *Numbering* 스킴은 XML 문서의 트리구조에서 루트부터 시작하여 깊이우선순회방법을 사용하여 숫자를 정의한다. 다른 방법으로는 *left-right*순회방법을 사용하여 *Numbering* 스킴을 적용하기도 한다[5].

표 1. XML 문서

```
<book>
  <title> xml </title>
  <Ilauthors>
    <author> jane </author>
    <author> john </author>
  </allauthors>
  <year> 2000 </year>
  <chapter>
    <head>Origins </head>
    <section>
      <head> kyungnam </head>
      <section>12</section>
      </section>
      <section> 1 </section>
    </chapter>
    <chapter>
      <head>DB </head>
      <section>
        <head> SQL</head>
        <section>22</section>
      </section>
      <section> 2 </section>
    </chapter>
</book>
```

XML 문서에 깊이우선순회 *Numbering* 스킴을 사용한 방법은 몇 가지로 구분되는데 그 중 하나는 0, 1, 2, 3, 4....과 같이 순차적인 숫자를 이용한 방법이고[9], 또 다른 방법으로는 순차적이면서 개체사이에 일정한 구간을 두는 방법이 있다[5]. *Numbering* 스킴를 살펴보면(*order, size*)의 쌍으로 구성되는데 *order*는 개체의 순서(*DFS*)를 의미하며 *size*는 개체의 특정 범위를 나타낸다. *order*와 *size*의 범위는 x가 y의 조상일 때 $order(x) < order\ (y) \leq order(x) + size(x)$와 같은 규칙이 적용된다. 또한 구간이 있는 경우는 개체사이의 일정한 범위까지 인덱스를 삽입할 수 있지만 범위를 벗어나면 많은 갱신비용과 오버헤드를 요구하게 되며 개체간의 *Parent-Child*의 관계가 뚜렷하게 나타나지 않은 단점이 있다.

XML 인덱스를 정의하기 위한 또 다른 방법으로는 *Bit Structured Schema* 방법이 있다[1].

2.2 XML 질의 처리

XML 질의 처리를 위해 깊이우선순회 *Numbering* 스킴을 이용해서 정의된 인덱스를 가지고 XML 데이터의 계층적인 구조를 검색하는 방법은 여러 가지가 있다. XML 문서의 계층적인 구조의 특성은 4 가지로 구분할 수 있다[9]. *Containment Property*는 두 개체의 관계가 *Ancestor-Descendant*인 경우를 의미한다. *Direct Containment Property*는 두 개체의 관계가 *Parent-Child*인 경우를 의미하는데 *Containment Property*이면서 레벨차이가 1인 경우이다. *Tight Containment Property*는 두 개체의 관계가 *Direct Containment Property*이면서 *Descendant*가 오직 하나만 있을 경우이다. 즉 *Parent-Child*관계이면서 하나의 *Child*를 가진 것을 의미한다. *Proximity property*는 두 개체사이의 거리가 일정한 범위내에 있는 것을 의미한다.

계층구조에 대한 질의처리를 위해 여러 조인알고리즘이 개발되었다. *MPMGJN* 알고리즘은 그림 2와 같이 두 개의 리스트를 입력받아서 숫자를 비교하여 조인하는 알고리즘이며 *Standard merge join, Index nested-loop join*과 차이점을 보인다. 또 다른 합병조인 알고리즘으로는 *Tree-Merge Join*과 *Stack-Tree Join* 알고리즘이 있다[7]. *Stack-Tree Join* 알고리즘은 스택을 사용해서 개체간의 비교 연산을 줄일 수 있다.

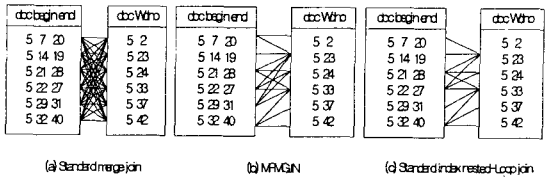


그림 2. merge join의 처리과정

3. XML 문서의 분할과 인덱스

3.1 독립적인 블록식별자를 가지는 경우

기존의 *Numbering* 스킴를 사용한 XML 문서의 인덱스 기법은 XML 문서에 갱신이 발생하게 되면 모든 인덱스를 재정의 해야 하는데, 이것은 상당한 처리비용을 요구하게 된다. 이러한 처리비용을 줄이기 위해서 문서 갱신시에 재정의 해야 하는 인덱스를 특정부분으로 제한하는 방법을 제안한다. XML 문서를 블록단위로 나누고 각 블록에 *Numbering* 스킴을 이용하여 인덱스를 정의하도록 한다. 따라서 XML 문서의 갱신시 해당 블록의 인덱스만 재정의 하도록 하여 인덱스 갱신에 따른 처리비용을 상당히 감소시킨다.

XML 문서를 블록단위로 분할한 후에도 개체사이의 계층구조를 유지해야 한다. 이 문제를 해결하기 위해서 블록과 블록사이의 관계를 두 개 이상 가지도록 하면 안되며 블록과 블록사이는 많아야 하나의 관계만을 갖도록 해야 한다. 그리고 각 블록마다 *Numbering* 스킴을 적용한다. 따라서 XML 문서의 계층구조는 블록사이의 계층구조와 각 블록의 *Numbering* 스킴을 이용하면 재건할 수 있다.

XML 문서를 블록 단위로 분할하기 위해서는 SAX 파서를 이용하거나 XML 문서를 처음부터 읽어 가면서 블록을 분할하여 블록단위로 *Numbering* 스킴을 적용한다.

BSA(*Block Split Algorithm*)은 XML 문서를 블록으로 나누고 블록단위로 *Numbering* 스킴을 적용하여 인덱스를 정의하는 알고리즘이다. 표 3은 *BSA*을 나타낸 것으로 *BSA*은 XML 문서와 정수 *N*을 입력받는다. 여기서 *N*은 한 블록이 가질 수 있는 최대 개체 수이다.

*BSA*에 의해서 생성된 개체는 아래와 같이 *ELEMENT*와 *TEXTS*의 형태를 가진다. *Docno*는 XML 문서번호이며, *Object name*은 XML 문서의 개체 이

름이며, *Block ID*는 각 블록을 구별하기 위한 블록식 별자이다.

ELEMENT(Docno, Object name, Block ID, level, begin:end, Parent Block, Parent begin:end)

TEXTS(Docno, Object name, Block ID, level, Wordno, Parent Block, Parent begin:end)

*level*은 블록에서 개체의 레벨을 나타내며, *Begin:end*는 블록에서 각 개체의 시작위치와 끝위치를 나타낸다. *Wordno*는 *TEXTS*개체의 위치이다. *Parent Block*은 블록과 블록사이의 관계에서 부모 블록을 의미하며, *Parent begin:end*은 부모블록에서 직접 관계가 있는 개체의 시작위치와 끝위치를 나타낸다. 그림 3과 표 2는 표 1의 XML 문서에 N을 6으로 하여 BSA를 적용한 경우의 트리구조와 인덱스이다.

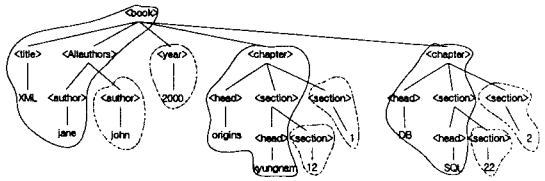


그림 3. 표 1의 트리 구조

3.1.1 XML 문서의 갱신에 따른 블록의 변화

블록화된 XML 문서에서 삽입, 삭제, 갱신 등의 작업이 일어날 경우의 블록구조를 살펴보면 다음과 같다.

① 삽입 : 가득찬 블록과 가득찬 블록사이의 삽입 시에는 새로운 블록을 생성하여 블록사이의 관계를

표 2. 표 1의 XML 문서를 BSA에 적용한 인덱스

Object name	No	Block ID	level	begin:end	Parent Block	Parentbegin:Parentend
<book>	1	0	0	0:9	null	null
<title>	2	0	1	1:3	null	null
XML	3	0	2	2	null	null
<allauthors>	4	0	1	4:8	null	null
<author>	5	0	2	5:7	null	null
jane	6	0	3	6	null	null
<author>	7	1	0	0:2	0	4:8
john	8	1	1	1	0	4:8
<year>	9	2	0	0:2	0	0:9
2000	10	2	1	1	0	0:9
<chapter>	11	3	0	0:9	0	0:9
<head>	12	3	1	1:3	0	0:9
Origins	13	3	2	2	0	0:9
<section>	14	3	1	4:8	0	0:9
<head>	15	3	2	5:7	0	0:9
kyungnam	16	3	3	6	0	0:9
<section>	17	4	0	0:2	3	4:8
12	18	4	1	1	3	4:8
<section>	19	5	0	0:2	3	0:9
1	20	5	1	1	3	0:9
<chapter>	21	6	0	0:9	0	0:9
<head>	22	6	1	1:3	0	0:9
DB	23	6	2	2	0	0:9
<section>	24	6	1	4:8	0	0:9
<head>	25	6	2	5:7	0	0:9
SQL	26	6	3	6	0	0:9
<section>	27	7	0	0:2	6	4:8
22	28	7	1	1	6	4:8
<section>	29	8	0	0:2	6	0:9
2	30	8	1	1	6	0:9

표 3. Block Split Algorithm

```
product BSA(XMLDocument XDoc , DesireNumber N)
begin
1 :  Locallevel Checklevel= 1;   // XML문서에서 개체 level
2 :  DesireNumber K= 1;          // N까지의 갯수를 제어
3 :  ObjectPoint LocalPoint;     // 개체의 위치변수 (시작시 처음위치를 나타냄)
4 :  Objectlevel Firstlevel=0;   // Block의 처음개체 level을 저장(실제 보다 1적다)
5 :  NumberingValue Number= 1;  // Block당 Numbering 하기 위한 값
6 :  BlockrObjectBalance ObjectBalanceCheck=0;  // Block당 Element 시작과 끝의 갯수 Check
7 :  BlockLevel BlockLevelNumber    // Block당 Level
8 :  LocalPoint=XDoc.FirstObject      // LocalPoint가 처음의 Object를 가려 치도록 함.
9 :
10 : while (XDoc의 끝이 아니면){
11 :          new 블록의 설정;
12 :          Firstlevel=Checklevel;
13 :          BlockLevelValuel=-1;
14 :    do{
15 :      if (LocalPoint가 가리키는 Object가 시작 Element일 경우){
16 :          Checklevel++;  BlockLevelValue++;
17 :          Number++;
18 :          K++;
19 :          ObjectBalanceCheck++;
20 :      LocalPoint가 가리키는 Object를 출력(Block IDr, ObjectName, Checklevel(BlockLevelValuel), Number(BeginNumber));
21 :      }
22 :
23 :       if ( LocalPoint가 가리키는 Object가 Text일 경우){
24 :          Checklevel++;  BlockLevelValue++;
25 :          Number++;
26 :          K++;
27 :      LocalPoint가 가리키는 Object를 출력(Block IDr, ObjectName, Checklevel(BlockLevelValuel), Number(BeginNumber));
28 :          Checklevel--;
29 :       }
30 :
31 :       if (LocalPoint가 가리키는 Object가 닫는 Element Object){
32 :            Checklevel--;  BlockLevelValue--;
33 :            ObjectBalanceCheck--;
34 :            Number++;
35 :          if (K≠0) and (ObjectBalanceCheck >= 0)   //Block당  태그의 시작과 끝의 Balance 체크
36 :              ObjectName이 같은 것 중에 가장 최근의 레코드부터 Number(EndValue)없는 레코드에 삽입;
37 :
38 :       }
39 :       LocalPoint =LocalPoint ->Next;
40 :    } while ((K < N-1) and (Checklevel > Firstlevel))
41 :
42 :         while(LocalPoint가 가리키는 Object가 닫는 Element Object)
43 :           Checklevel--;
44 :           LocalPoint =LocalPoint ->Next;
45 :         }
46 :
47 :     Element의 End Value 값이 없는 것을 찾아 나중에 생성된 것부터 다음 Number값 삽입(all);
48 :     Firstlevel의 개체의 level보다 한 단계 적은 최근의 개체의 위치를 부모로 생성한다(처음 block 제외);
49 :   K=-1;
50 :   Number=-1;
51 :   ObjectBalanceCheck=0;
52 : }
    end
```

표현하면 된다. 삽입하고자 하는 블록사이에 가득 차지 않은 블록이 존재하면 그 블록에 개체를 삽입하고 해당 블록의 인덱스를 재정의 하면 된다.

② 삭제 : 삭제의 경우는 삭제하고자 하는 개체를 삭제하고 그 개체의 모든 자손들을 삭제해야 한다.

③ 갱신 : 해당 개체를 갱신하면 된다.

따라서 XML 문서가 변경될 경우 최악의 경우는 해당 블록의 바로 다음 블록의 인덱스만 재정의 하면 된다. 이와 같이 XML 문서를 블록단위로 나누어 인덱스를 정의할 경우 XML 문서 변경 시에 인덱스를

재정의 하는 비용을 줄일 수 있다.

3.2 블록식별자가 부모 블록의 정보를 가지는 경우

*Ancestor-Descendant*관계를 비교할 때 *Descendant*개체가 포함된 블록의 처음개체와 *Ancestor-Descendant*관계가 성립하면 *Descendant*의 블록에 있는 모든 개체가 *Ancestor-Descendant*관계가 성립하며 반대의 경우도 마찬가지이다.

또한 블록과 블록이 *Ancestor-Descendant*관계가 아니면 포함된 모든 개체 또한 *Ancestor-Descendant*관계가 성립하지 않는다. 따라서 *Block ID*가 조상블록의 정보를 포함한다면 *Ancestor-Descendant*관계에 대한 검색 성능을 크게 향상시킬 수 있다. 이 경우 개체 삽입시 모든 자손의 *Block ID*를 갱신해야 하는 단점이 있다. 그러나 모든 개체의 트리구조보다는 블록단위의 트리구조가 노드 수에서 상당히 적다. 예를 들어 N=6 일때 개체 수가 100이면 XML 트리는 100개의 노드를 가지지만 모든 블록이 가득찼을 경우 블록의 수는 100/6=16.66이므로 블록의 트리 구조에서는 17개정도의 노드가 생성된다. 따라서 개체의 관계 검색 시 블록의 관계를 먼저 계산함으로써 개체 하나 하나를 비교하는 것보다 빠르게 이들 관계를 검색할 수 있다. 또한 *Block ID*정보를 별도로 관리할 경우 갱신에 따른 인덱스 재정의 오버헤드를 상당히 줄일 수 있다.

그림 4를 블록단위의 트리구조로 나타내면 그림 5와 같다. 그림 5에서 *Block ID*만으로 블록의 관계를

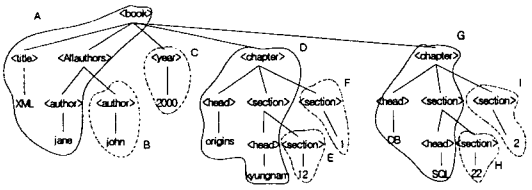


그림 4. 표 1의 트리 구조

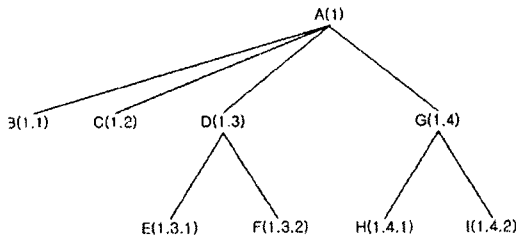


그림 5. 블록트리

알 수 있다. 예를 들어 E(1.3.1)은 1.3 블록의 *Child*이며 1 블록의 *Descendent*를 의미한다. 또한 1.3 블록이 1 블록의 *Child*인 것도 암시한다.

인덱스 개체와는 별도로 *Block ID* 정보를 관리한다면 XML 문서에 갱신이 발생할 경우 *Block ID*와 해당 개체의 인덱스 정보만 재정의하면 된다. 따라서 개체의 인덱스 구조를 나타내는 객체데이터테이블과 블록의 관계를 나타내는 블록관계테이블로 구분하여 표현한다. 표 4는 각각 객체데이터테이블과 블록관계테이블을 보여주고 있다. 블록관계테이블에서 블록의 관계를 쉽게 확인할 수 있다.

표 4. 그림 14의 분할된 인덱스

Object name	No	Block Number	level	begin:end
<book>	1	0	0	0:9
<title>	2	0	1	1:3
XML	3	0	2	2
<allauthors>	4	0	1	4:8
<author>	5	0	2	5:7
jane	6	0	3	6
<author>	7	1	0	0:2
john	8	1	1	1
<year>	9	2	0	0:2
2000	10	2	1	1
<chapter>	11	3	0	0:9
<head>	12	3	1	1:3
Origins	13	3	2	2
<section>	14	3	1	4:8
<head>	15	3	2	5:7
kyungnam	16	3	3	6
<section>	17	4	0	0:2
12	18	4	1	1
<section>	19	5	0	0:2
1	20	5	1	1
<chapter>	21	6	0	0:9
<head>	22	6	1	1:3
DB	23	6	2	2
<section>	24	6	1	4:8
<head>	25	6	2	5:7
SQL	26	6	3	6
<section>	27	7	0	0:2
22	28	7	1	1
<section>	29	8	0	0:2
2	30	8	1	1

표 4. 계속

Block Number	Block ID	Parentbegin: Parentend
0	1	NULL
1	1.1	4:8
2	1.2	0:9
3	1.3	4:8
4	1.3.1	4:8
5	1.3.2	0:9
6	1.4	0:9
7	1.4.1	4:8
8	1.4.2	0:9

3.2.1 XML문서의 갱신에 따른 블록의 변화

XML 데이터 삽입, 삭제 및 갱신에 따른 블록트리 구조와 블록관계테이블의 변화를 살펴보자.

① 삽입 : 가득찬 블록과 가득찬 블록사이의 삽입 시에는 새로운 블록을 생성하여 블록사이의 관계를 표현하면 된다. 삽입하고자 하는 블록사이에 가득 차지 않은 블록이 존재하면 그 블록에 개체를 삽입하고 블록관계테이블 재정의 하면 된다. 예를 들어 그림 4에서 *A*블록과 *D*블록은 가득 찬 블록이다. 따라서 개체가 삽입될 경우 블록트리의 구조는 그림 6과 같이 된다. 블록관계테이블은 표 4에서 표 5와 같이 변화하며 인덱스 데이터 테이블에 *NEW*블록의 데이터를 삽입하면 된다.

② 삭제 : 삭제의 경우는 개체를 삭제하고 그 개체의 모든 자손들을 삭제해야 한다. 그리고 개체가 없는 블록은 삭제한다.

③ 갱신 : 해당 개체를 갱신하면 된다.

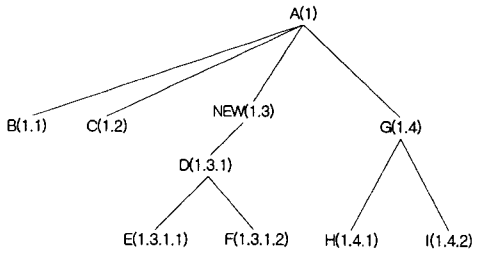


그림 6. 블록트리

4. XML 질의 처리

4.1 독립적인 블록식별자를 가지는 경우

*Parent-Child Block Merge Algorithm*은 정의된 인덱스를 이용하여 개체간의 관계가 *Parent-Child*인 개체를 검색하는 합병조인알고리즘이다. 같은 블록내에서 개체간의 *Parent-Child*인 관계의 검색은 정의된 인덱스의 *begin*과 *end, level, Block*을 비교하면 된다.

표 5. 블록관계 테이블

Block Number	Block ID	Parentbegin: Parentend
0	1	NULL
1	1.1	4:8
2	1.2	0:9
9	1.3	0:9
3	1.3.1	4:8
4	1.3.1.1	4:8
5	1.3.1.2	0:9
6	1.4	0:9
7	1.4.1	4:8
8	1.4.2	0:9

예를 들어 그림 7에서 *Docno*를 제외한 *<book>*과 *<title>*의 정의된 인덱스을 보면 *<book>*은 *(<book>, 0, 0, 0:9, null, null)*이고, *<title>*은 *(<title>, 0, 1, 1:3, null, null)*이다. *<book>*과 *<title>*의 관계가 *Parent-Child*관계인지를 비교하려면, 먼저 *Block ID*를 비교하고 다음으로 *begin:end*를 비교한 후 마지막으로 *Level*을 비교하면 된다. *<book>*과 *<title>*의 *Block ID*가 같고 *begin:end*에서 *<title>*이 *<book>*에 포함되고 레벨의 차이가 1이기 때문에 *Parent- Child* 관계가 성립함을 알 수 있다. 그림 7에서 *<allauthors>*와 *<author>*는 다른 블록의 개체사이의 관계가 *Parent-Child*인 경우이다. 각각의 인덱스는 *(<allauthors>, 0, 1, 4:8, null, null), (<author>, 1, 0, 0:2, 0, 4:8)*이다. 다른 블록의 개체사이가 *Parent-Child*관계가 성립하려면 *Parent*개체의 *begin:end, Block ID*와 *Child*개체의 *ParentBegin:ParenteEnd,*

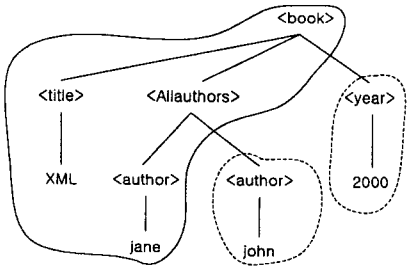


그림 7. XML문서의 블록도

Parent Block ID 값이 각각 같고, *Child*개체의 *begin*이 0이면 된다. 따라서 블록이 다른 경우는 오직 하나의 *Parent-Child*관계만 존재한다.

*Ancestor-Descendant Block Merge Algorithm*은 정의된 인덱스를 이용하여 개체간의 관계가 *Ancestor-Descendant*인 개체를 검색하는 합병조인알고리즘이다. 같은 블록에서 개체간의 *Ancestor-Descendant*인 관계를 검색하기 위해서는 인덱스의 *begin:end, Block*을 비교하면 된다. 예를 들어 그림 7에서 *<book>*은 *(<book>, 0, 0, 0:9, null, null)*이고 XML은 *(XML, 0, 2, 2, null, null)*이다. *Block ID*를 비교하고 다음으로 *begin:end*을 비교한다. $0 < 2$ 이고 $2 < 9$이므로 *XML*은 *<book>*에 포함된다.

그림 7에서 *<book>*과 *<author>*는 다른 블록의 개체사이의 관계가 *Ancestor-Descendant*인 경우이다. 개체들의 인덱스는 *(<book>, 0, 0, 0:9, null, null)*와 *(<author>, 1, 0, 0:2, 0, 4:8)*이다. 두 개체의 관계를 검색하기 위해서 자손의 *Parent Block, ParentBegin:ParenteEnd* 값으로 순환적으로 조상을 찾는다. 이러한 반복적인 작업은 검색하고자 하는 조상과 같은 *Block ID* 값이 나오거나 *null*일 때 끝난다. *Block ID*값이 나오지 않으면 두 개체는 *Ancestor-Descendant* 관계가 성립하지 않으며, 조상의 *Block ID*가 나왔을 경우는 서로의 *begin:end* 값을 비교해서 포함 관계가 성립하면 *Ancestor-Descendant* 관계이다. 표 6은 *Ancestor-Descendant Block Merge Algorithm*을 나타낸 것이다. *Ancestor-Descendant* 경우는 *Parent-Child* 보다는 많은 검색 비용을 요구한다.

4.2 블록식별자가 부모 블록의 정보를 가지는 경우

블록식별자가 부모 블록의 정보를 가지는 경우 같은 블록에서 개체간의 *Parent-Child*인 관계를 검색하기 위해서는 정의된 인덱스의 *begin:end, level, Block*을 비교하면 된다. 다른 블록의 개체간의 *Parent-Child* 관계 검색의 경우는 블록과 블록의 관계가 많아야 하나만 존재하므로 *Parent-Child* 관계인 개체가 검색되었을 경우 다음 블록을 검색하게 된다.

표 7은 블록식별자가 부모블록정보를 가지는 경

표 6. Ancestor-Descendant Block Merge Algorithm

```
Procedure ADBMA(Alist, Dlist)
Begin
1 :     set A-cursor beginning of Alist1
2 :     set D-cursor beginning of Dlist2
3 :
4 :     for(A-cursor=Alist->firstNode : A-cursor !=NULL :   A-cursor =A-cursor ->NextNode)
5 :     {
6 :        for(D-cursor=Dlist->firstNode : D-cursor !=NULL :   D-cursor =D-cursor ->NextNode)
7 :             if (A-cursor.Block ID == D-cursor.Block ID)
8 :                {
9 :                    if (A-cursor.begin < D-cursor.begin) && (A-cursor.end > D-cursor.end)
10:                       OutputList;
11:
12:                } else if {
13:
14:                    if (A-cursor == ParentBlock(D-cursor)      // 조상과 자손 관계 확인
15:                      {   MarkID = D-cursor.Parent Block ID;
16:                          while(A-cursor.Block ID == MarkID)
17:                          {
18:                           OutputList;
19:                           D-cursor=Dlist->firstNode;
20:                          }
21:                      } else if {
22:                                D-cursor=Dlist->firstNode;
23:                               }
24:                }
25:    }
end
```

표 7. 부모 블록 정보를 가지는 경우의 Parent-Child Block Merge Algorithm

```
procedure PCBMA (ListOne, ListTwo)
begin
01: set CursOne to be the first object of ListOne;
02: set CursTwo to be the first object of ListTwo;
03: mark = CursTwo;
04: while ((CursOne ≠ end of ListOne) and ( CursTwo ≠ end of ListTwo)) begin
05:   if (CursTwo = end of ListTwo)
06:     begin
07:        CursOne ++;
08:        CursTwo=mark;
09:     end
10:     else
11:     begin
12:       if (CursOne.blockid = CursTwo.blockid)
13:       begin
14:            if ((CursOne contains CursTwo)and (CursOne.level == (CursTwo.level-1)))
15:            begin
16:                  mark = CursTwo;
17:                  do
18:                    merge CursOne and CursTwo;
19:                    CursTwo++;
20:                  while ((CursOne.blockid = CursTwo.blockid) and (CursOne contains CursTwo)
21:                        and (CursOne.level == (CursTwo.level-1)))
22:                  CursOne++;
23:                  CursTwo = mark;
24:            end
25:       end
26:       else
27:       begin
28:        if (CursOne.blockid <> CursTwo.blockid)
29:        begin
30:         if (CursOne.blockid contains CursTwo.blockid)
31:         begin
32:                   Find the following block of CursOne.block in the path from CursOne.block to
                      CursTwo.block.;
33:              set the there block CursThree;
34:                    if (CursThree.parentbegin == CursOne.begin) and (CursThree.parentend ==
                      CursOne.end)
35:                and (CursThree == CursTwo)
36:              begin
37:                   mark = CursTwo;
38:               do
39:                   merge CursOne and CursTwo;
40:                   Blockmake= CursTwo.blockid;
41:                   CursTwo++;
42:                     while (Blockmake== CursTwo.blockid)
43:                    begin
44:                         CursTwo++;
45:                    end
46:                     Find the following block of CursOne.block in the path from CursOne.block to
                        CursTwo.block.
47:                  set the there block CursThree
48:                  while ((CursOne.blockid <> CursTwo.blocki) and (CursOne.blockid contains
                     CursTwo.blockid) and (CursThree.parentbegin == CursOne.begin) and
                     (CursThree.parentend == CursOne.end) and (CursThree == CursTwo))
49:                   CursOne++;
50:                   CursTwo = mark;
51:                 end
52:              end
53:             end
54:           end
55:       end
56:   end
57: end
```

표 8. 부모 블록 정보를 가지는 경우의 Ancestor-Descendant Block Merge Algorithm

```
procedure ADBMA (ListOne, ListTwo)
begin
01:  set CursOne to be the first object of ListOne;
02:  set CursTwo to be the first object of ListTwo;
03:  mark = CursTwo;
04:  while ((CursOne ≠ end of ListOne) and ( CursTwo ≠ of ListTwo)) begin
05:    if (CursTwo= end of ListTwo)
06:    begin
07:           CursOne ++;
08:           CursTwo=mark;
09:    end
10:    else
11:    begin
12:      if (CursOne.blockid = CursTwo.blockid)
13:      begin
14:        if (CursOne contains CursTwo)
15:         begin
16:                mark = CursTwo;
17:                do
18:                   merge CursOne and CursTwo;
19:                   CursTwo++;
20:                while ((CursOne contains CursTwo) and (CursOne contains CursTwo))
21:                CursOne++;
22:                CursTwo = mark;
23:         end
24:      end
25:      else
26:      begin
27:        if (CursOne.blockid <> CursTwo.blockid)
28:        begin
29:          if (CursOne.blockid contains CursTwo.blockid)
30:          begin
31:                Find the following block of CursOne.block in the path from CursOne.block to
                   CursTwo.block.;
32:             set the there block CursThree;
33:             if (CursThree.parentbegin >= CursOne.begin) and (CursThree.parentend <= CursOne.end)
34:             begin
35:                mark = CursTwo;
36:                do
37:                   merge CursOne and CursTwo;
38:                   Blockmake= CursTwo.blockid;
39:                   CursTwo++;
40:                   while (Blockmake== CursTwo.blockid)
41:                   begin
42:                      merge CursOne and CursTwo;
43:                      CursTwo++;
44:                   end
45:                    Find the following block of CursOne.block in the path from CursOne.block to
                       CursTwo.block.;
46:                   set the there block CursThree;
47:                while ((CursOne.blockid <> CursTwo.blocki) and (CursOne.blockid contains
                   Curs Two.blockid) and (CursThree.parentbegin >= CursOne.begin) and (CursThree.
                   parentend <= CursOne.end))
48:                CursOne++
49:                CursTwo = mark;
50:             end
51:          end
52:        end
53:      end
54:    end
55:  end
56: end
```

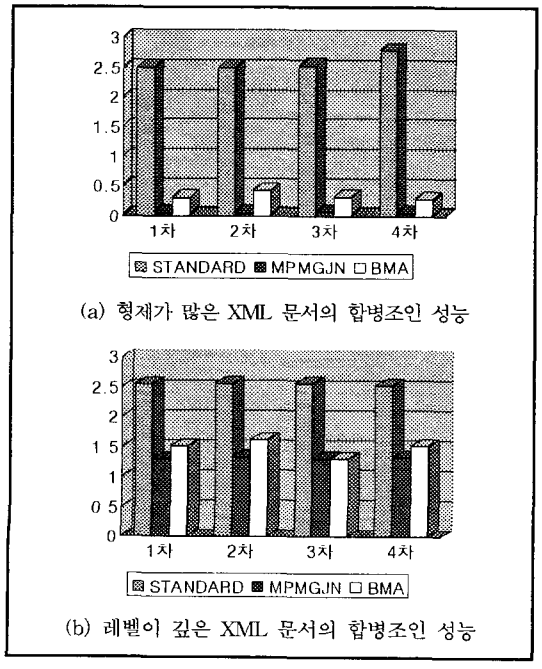
우의 *Parent-Child Block Merge Algorithm*을 보여주고 있다.

표 8의 알고리즘은 블록식별자가 부모 블록의 정보를 가지는 경우의 *Ancestor-Descendant Block Merge Algorithm*이다. 같은 블록내의 개체들의 관계를 검색하기 위해서는 정의된 인덱스의 *begin:end, Block*을 비교하면 된다. 그림 4와 그림 5에서 *A*블록의 <*book*>과 *H*블록의 <*section*>관계는 다른 블록의 개체사이의 관계이다. *A*블럭의 <*book*>의 *Block ID* 값은 1이고 *H*블록의 <*section*>의 *Block ID* 값이 1.4.1이므로 이들 블록이 서로 관계를 가진다는 것을 알 수 있다. 다음으로 블록관계테이블에서 *Block ID*의 값이 1.4인 *Parentbegin: Parentend*값을 <*book*>의 *begin:end*값과 비교한다. 값이 같거나 포함되면 이들의 관계는 *Ancestor-Descendant* 관계이다. 이처럼 블록관계테이블을 이용하여 개체의 관계를 파악할 수 있다. 따라서 블록이 다를 경우에는 블록사이의 관계만으로 블록에 정의된 모든 개체의 관계를 알 수 있다.

5. 성 능

표 9는 합병조인연산의 실험결과를 보여주고 있다. 실험에 사용된 XML 문서의 개체수는 약 3000개인데 레벨이 깊은 경우와 형제가 많은 두 유형의 XML 문서를 대상으로 하였다. 본 논문에서 제안한 *BMA*는 표준조인방법보다는 성능이 우수하고 *MPMGJN*과 유사한 성능을 보여주고 있다.

표 9. 합병조인연산의 실험결과



(a) 형제가 많은 XML 문서의 합병조인 성능

(b) 레벨이 깊은 XML 문서의 합병조인 성능

6. 결 론

본 논문에서는 XML문서의 저장과 질의 처리를 위해 역 인덱스 기법을 제안하였다. *BSA(Block Split Algorithm)*에 의해서 XML 문서를 블록단위로 나누고 각 블록내에서는 *Numbering* 스킴을 이용하여 인덱스를 정의하도록 한다. XML 문서의 계층구조를 그대로 유지하기 위해서 블록으로 나눌때 블록과 블록사이는 많아야 하나의 관계만을 가지도록 해야 한다. 이렇게 만들어진 인덱스는 XML 문서의 계층구조 정보를 유지하면서 XML 문서의 변경이 발생하게 되면 해당 블록의 인덱스만 재정의 하도록 하여 인덱스 재정의에 따른 부담을 줄여준다.

또한 XML 질의 처리를 위해서 *Parent-Child Block Merge Algorithm*과 *Ancestor-Descendent Block Merge Algorithm*을 제안하였다. *Parent-Child Block Merge Algorithm*은 *Tree-Merge Join*과 *MPMGJN*과 비슷한 성능을 가진다. *Ancestor-Descendent Block Merge Algorithm*에서 *Descendant*개체가 포함된 블록의 처음개체와 *Ancestor-Descendant*관계가 성립하면 *Descendant*의 블록에 있는 모든 개체가 *Ancestor-Descendant*관계가 성립한다. 이러한 성질을 이용하여 검색과정에서 필요하지 않은 연산을 제거한다.

그리고 질의 처리에 있어 *Ancestor-Descendent* 관계의 효율적인 검색을 위해서 블록식별자가 조상블록의 정보를 유지하는 방법을 제안하였다. 이 방법은 XML 데이터 삽입시 모든 자손의 블록식별자를 갱신해야 하는 단점이 있다. 그러나 모든 개체의 트리구조보다는 블록단위의 트리구조가 노드 수에서 적다. 따라서 개체의 관계 검색시 블록의 관계를 먼저 검사함으로써 개체를 하나 하나 비교하는 것보다 빠르게 이들 관계를 검색할 수 있으며 블록식별자정보를 별도로 관리함으로써 인덱스 재정의 시 발생하는 오버헤드를 줄일 수 있다.

참 고 문 헌

[1] V. Aguil'era, S. Cluet, P. Veltri, D. Vodislav, F Wattez. "Querying xml documents in xyleme". *Proceedings of the ACM-SIGIR 2000 Workshop on XML and Information Retrieval.*

[2] D. Chamberlin, D. Florescu, J. Robie, J. Simeon, and M. Stefanescu. "Xquery A query language for XML". *W3C Working Draft.* Available from http://www.w3.org/TR/xquery, Feb 2001.

[3] S.Y. Chien, V.J. Tsotras, and C. Zaniolo. "Version Management of XML Documents". *Proceedings of International Workshop on the Web and Databases, WebDb* 2000.

[4] D. D. Kha, M. Yoshikawa, and S. Uemura. "An XML indexing structure with relative region coordinate". *Proceedings of 17th International Conference on Data Engineering,* 2001.

[5] Q. Li, B. Moon. "Indexing and querying XML data for regular path expressions". *Proceedings of the 27th VLDB Conference,* 2001.

[6] F. Rizzolo, A. Mendelzon. "Indexing XML Data with ToXin". *Proceedings of Fourth International Workshop on the Web and Databases, WebDb* 2001.

[7] D. Srivastava, S. Al-Khalifa, H.V. Jagadish, N. Koudas, J.M. Patel, Y.Wu. "Structural Joins: A Primitive for Efficient XML Query Pattern Matching". *Proceedings of ICDE* 2002.

[8] F. Tian, D. DeWitt, J. Chen, and C. Zhang. "The Design and Performance Evaluation of Alternative XML Storage Strategies". *Technical report, CS Dept., Universiy of Wisconsin, 2000.*

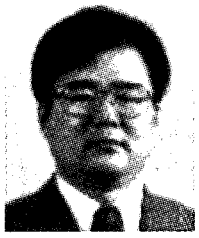
[9] C. Zhang, J. Naughton, D. DeWitt, Q. Luo, G Lohman. "On Supporting Containment Queries in Relational Database Management Systems". *Proceedings of ACM SIGMOD Conference on Management of Data* 2001.

[10] W3C Recommendation. XML Path Language (Xpath) 1.0. In *http://www.w3.org/TR/xpath.* 1999.

김 종 명

2001년 경남대학교 전자계산학과 학사
2003년 경남대학교 컴퓨터공학과 석사
2003년~삼창기업 제어계측연구소 연구원

관심분야 : 데이타베이스, 분산처리, 데이타 모델링, Network protocol



진 민

1982년 서울대학교 계산통계학과 학사
1984년 한국과학기술원 전산학과 석사
1997년 University of Connecticut 컴퓨터공학과 박사
1985년~현재 경남대학교 정보통신공학부 교수

관심분야 : 데이타베이스, 객체지향데이타베이스, 웹데이타베이스, XML, 데이타 모델링

교 신 저 자

김 종 명 631-701 경남 마산시 월영동 449 경남대학교 컴퓨터공학과