XML 데이터베이스에서 효율적인 경로처리를 위한 구조적 세미조인 기법

(Structural Semi-Join Operators for Efficient Path Processing in XML Databases)

손 석 현 †　　신 효 섭 ††

(Seokhyun Son)　　(Hyoseop Shin)

요 약 구조적 조인은 효율적인 XML 질의 처리를 위한 핵심 연산자 중의 하나이다. 구조적 조인은 대용량의 XML 노드들을 대상으로 계층관계(조상-자손 및 부모-자식관계)를 형성하는 쌍을 효율적으로 계산한다는 측면에서, 경로패턴으로 표현된 질의를 처리하는 데 주로 사용될 수 있다. 하지만 구조적 조인 알고리즘은 XML의 경로 처리 과정에서 많은 오버헤드를 야기 시킨다. 이에 대한 개선된 연산자인 구조적 세미조인은 효율적인 처리를 위하여 XML 노드 간의 조인 결과를 조상노드 혹은 자손노드로 한정시키는 새로운 연산자이다. 본 논문에서는 구조적 세미조인 알고리즘을 소개하고, 구조적 세미조인을 이용한 경로처리 알고리즘을 제시한다. 실험을 통하여 개선된 방식의 구조적 세미조인 알고리즘이 XML 경로처리에 있어서 매우 효율적임을 보여준다.

키워드 : XML, 구조적 조인, 구조적 세미조인, 경로처리

Abstract The structural join is one of core operators for efficient processing of XML queries. It can be mainly used for path-represented XML queries as it efficiently retrieves the node pairs that form a hierarchical relationship (i.e., ancestor-descendant, parent-child relationship) among large-scale XML nodes. However, the structural join algorithms still suffer potential overhead in the middle of processing of XML path queries. In addressing this problem, the structural semi-join is proposed as a novel operator that retrieves only the ancestor or descendant nodes as join results for efficient processing. In this paper, we describe the algorithms for the structural semi-join and present the methods of XML path processing based on the structural semi-join algorithms. The experimental results show that the structural semi-join algorithms are very efficient in processing XML path processing.

Key words : XML, Structural Join, Structural Semi-Join, XML path processing

· 이 논문은 제 34회 추계학술대회에서 'XML 데이터베이스에서 효율적인 경로처리를 위한 구조적 세미조인 기법'의 제목으로 발표된 논문을 확장한 것임

† 학생회원 : 건국대학교 신기술융합학과
seokhyunson@gmail.com

†† 종신회원 : 건국대학교 신기술융합학과 교수
hsshin@konkuk.ac.kr

논문접수 : 2008년 1월 8일
심사완료 : 2009년 11월 24일

Copyright©2010 한국정보과학회 : 개인 목적이나 교육 목적인 경우, 이 저작물의 전체 또는 일부에 대한 복사본 혹은 디지털 사본의 제작을 허가합니다. 이 때, 사본은 상업적 수단으로 사용할 수 없으며 첫 페이지에 본 문구와 출처를 반드시 명시해야 합니다. 이 외의 목적으로 복제, 배포, 출판, 전송 등 모든 유형의 사용행위를 하는 경우에 대하여는 사전에 허가를 얻고 비용을 지불해야 합니다.

정보과학회논문지: 컴퓨팅의 실제 및 레터 제16권 제2호(2010.2)

1. 서 론

XML 질의 처리시, XML노드 간의 조상-자손(ancestor-descendant) 과 부모-자식(parent-child) 간의 관계는 중요한 이슈이다. 그 이유로는 XML 질의의 경로 표현식(path expression)은 각각의 구조적 관계의 조합이 중요한 역할을 차지하고 있기 때문이다. 예를 들면, XPath 질의 중, *"Paper[Title='XML']/Author"*는 *'Paper'*와 *'Title'*, *'Paper'*와 *'Author'*같은 두 개의 부모-자식 간의 관계가 포함되어 있다. XML 데이터를 트리 구조로 표현할 때, XML노드 간의 관계는 숫자로 나타낼 수 있는데, 각각의 노드는 (*docid, start_pos, end_pos, level*)와 같은 정보로 표현된다. *docid*는 문서를 구별하는 정보이고, *start_pos*와 *end_pos*는 XML 노드의 시작하는 지점과 끝나는 지점을 나타낸다. 그리고 *level*은 root 노드로부터의 깊이(depth)를 나타낸다. 임의의 2개의 XML노드에 대해서 이와 같은 정보가 제공될 때에, 그들 간의 구조적 관계는 노드들의 속성을 비교하는 것으로 결정된다. 2개의 노드 r과 s가 있을 때, 그 둘의 관계가 조상-자손의 관계에 있다면 "($r.docid = s.docid$) and ($r.start_pos < s.start_pos$) 그리고 ($r.end_pos > s.end_pos$)" 의 조건식으로 표현된다. 특히, 부모-자식 관계일 때에는 "($r.level = s.level-1$)" 의 조건이 추가된다.

구조적 조인(Structural Join)[2]은 수많은 XML 노드 사이에 존재하는 이와 같은 구조적 관계를 해석하여 효율적으로 처리하는 방법을 최초로 제시하였다. 하지만 구조적 조인은 XML의 경로 패턴 처리 과정에서 불필요한 중간결과를 야기시킨다는 문제점을 가진다.

예를 들어서, 아래 그림 1과 같은 간단한 XML 문서 트리에 대하여 XPath 질의("*A//D*")를 구조적 조인을 통해서 처리하는 과정을 보자.

먼저, A와 D 노드에 대한 구조적 조인을 실행하면,

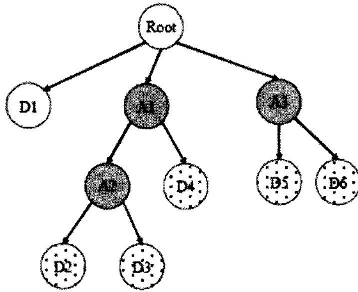

그림 1 간단한 XML문서 트리

{*<D2,A1>*,*<D2,A2>*, *<D3,A2>*, *<D4,A1>*, *<D5,A3>*, *<D6,A3>* } 이 중간 결과로 발생하고, 추가적인 프로젝트 연산을 통해서 A 노드들을 제거하면, {*<D2>*, *<D3>*, *<D4>*}의 최종 결과를 얻을 수 있다.

즉, XPath 질의에 대하여 옳은 결과를 얻기 위해서는, 구조적조인 연산만으로는 충분하지 못하므로 추가적인 프로젝트 연산이 요구된다. 질의 처리의 효율성 관점에서, 마지막 결과와 비교해서 구조적 조인의 중간 결과들은 상당한 부분이 중복될 수 있다. 게다가, 구조적 조인 연산자는 노드들이 쌍으로 나온다. 예를 들어, 위의 질의에서("*A//D*"), 구조적 조인의 결과에 포함된 A노드들은 질의의 결과에는 필요하지 않은 것들이다.

이 문제를 극복하기 위하여, 구조적 세미조인[1]은 XML 경로패턴을 효율적으로 처리하기 위하여 조인 결과를 조상노드 혹은 자손노드로 한정하여 반환하는 새로운 연산자로 제시되었다.

이에 본 논문에서는 구조적 세미조인을 이용한 효율적인 XML 경로패턴 처리 알고리즘을 제시하고, 대용량의 XML 데이터 집합에 대하여 실험을 통하여 그 성능을 분석한다.

본 논문의 구성은 다음과 같다. 2장에서는 구조적 세미조인의 정의 및 알고리즘을 소개한다. 3장에서는 구조적 세미조인을 이용한 경로처리 알고리즘을 제시한다. 4장은 실험결과를 제시한다. 5장은 이 논문의 결론을 제시한다.

2. 구조적 세미조인

정의 1(Structural Semi-Join) *start_pos* 순으로 정렬되어 있는 두 개의 노드 리스트 *AList*와 *DList*에 대하여 *StructuralSemiJoin(AList,DList)* 는 $(a \in AList) \cap (d \in DList) \cap (a.docid=d.docid) \cap (a.start_pos < d.start_pos) \cap (a.end_pos > d.end_pos)$ 을 만족하는 결과 노드(*<a>* 또는 *<d>*)로 구성된 집합을 반환하는 연산자로 정의한다. 단, 중복된 노드는 결과 리스트에 존재하지 않는다.

구조적 세미조인은 결과 반환에 따라, 두 가지로 분류된다.

정의 2 (Structural-SemiJoin-Desc) *Structural-SemiJoin-Desc(AList,DList)*는 *d.start_pos*의 정렬된 순으로 *<d>* 노드들을 단일 노드로 반환하는 구조적 조인 연산자이다.

정의 3 (Structural-SemiJoin-Anc) *Structural-SemiJoin-Desc(AList,DList)*는 *d.start_pos*의 정렬된 순으로 *<a>* 노드들을 단일 노드로 반환하는 구조적 조인 연산자이다.

2.1 스택기반의 구조적 세미조인

기존의 스택 기반의 구조적 조인 알고리즘[2]을 세미조인 버전으로 확장한 스택 기반의 구조적 세미조인 알고리즘인 Naïve-SSJoin을 먼저 설명한다. 자손 노드를 리턴하는 스택기반의 구조적 세미조인(Naïve-SSJoin-Desc)은 이전에 제시되었던 스택기반의 구조적 조인과 마찬가지로 조상 노드를 저장하는 스택을 가지고 있다. 스택에 있는 조상노드와 매치되는 자손노드가 들어왔을 때, 이전에 반환 되었던 노드가 아니였을 경우, 결과 값으로 반환한다. 조상 노드를 반환 하는 스택기반의 구조적 세미조인(Naïve-SSJoin-Anc) 역시 구조적 조인의 Stack-Tree-Anc 알고리즘과 마찬가지로 Self-list와 Inherit-list와 같은 대용량 연결리스트를 사용한다. 조상노드가 스택에서 팝이 일어날 경우, 모든 노드는 스택에서 리스트로 이동한다. 그리고, 매치되는 자손노드가 나타날 때, 그 노드가 이전에 반환된 적이 없었던 노드라면, 결과 값으로 반환한다.

2.2 NStack-SSJoin-Desc 알고리즘

2.2와 2.3 절에서는 구조적 세미조인의 개선된 버전인 NStack-SSJoin-Desc 및 NStack-SSJoin-Anc 알고리즘을 설명한다. NStack-SSJoin-Desc 알고리즘은 이전에 제시되었던 알고리즘과는 달리 스택을 사용하지 않고, 조상인디케이터(Ancestor Indicator)라는 메모리 변수를 스택 대용으로 사용하여 질의 처리 중 가장 큰 end_pos를 가진 조상노드만을 저장한다는 특징을 가진다. 이러한 조상인디케이터는 이전에 제시된 구조적 조인 알고리즘의 스택에 저장된 노드들 중 제일 하단과 일치하며, 스택에 들어오는 나머지 조상노드들은 인디케이터의 자손노드이므로 연산 과정에 필요하지 않게된다. 만약, 조상인디케이터가 널(null)값이 아니면, 들어오는 자손노드들은 결과값으로 반환한다. 그림 2는 NStack-SSJoin-Desc 알고리즘을 나타낸다. 알고리즘의 자세한 처리과정에 대한 예는 구조적 세미조인[1]을 참고하면 된다.

2.3 NList-SSJoin-Anc 알고리즘

Naïve-SSJoin-Anc알고리즘은 Self-list, Inherit-list

```
1   Algorithm NStack-SSJoin-Desc (AList, DList)
2   a = AList->firstNode; d = DList->firstNode; OutputList =
    NULL;
3   ancestor_indicator = NULL;
4   while ( (AList->isempty() == false) || (DList->isempty() ==
    false) || (ancestor_indicator != NULL) ) {
5      if ( (ancestor_indicator != NULL) &&(a.startpos >
    ancestor_indicator.endpos)          &&(d.startpos          >
    ancestor_indicator.endpos) )
6                       ancestor_indicator = NULL;
7      else if (a.startpos < d.startpos) {
8            if ( ancestor_indicator == NULL )
9                       ancestor_indicator = a;
10                      a = a->nextNode;
11           }
12           else {
13                      if ( ancestor_indicator != NULL )
14                      append d to OutputList;
15                      d = d->nextNode;
16           }
17  }
18  Return OutputList;
```

그림 2 NStack-SSJoin-Desc 알고리즘

와 같은 연결리스트를 사용하는데, 이러한 리스트들은 결과 값으로 리턴되는 조상노드를 기억하는데 사용한다. 이와는 달리 NStack-SSJoin-Anc은 조상노드들이 반환되는 과정에서 각각의 매치되는 자손노드를 기억할 필요는 없다는 점에 착안하여 알고리즘을 개선하였다. 즉, 현재 스택에 존재하는 조상노드들에 대하여 매치되는 자손노드가 발견되자마자, 스택에 있는 모든 조상노드를 반환하고, 스택을 비운다. 그로 인해 Self와 Inherit 리스트

```
1   Algorithm NList-SSJoin-Anc (AList, DList)
2   a=AList->firstNode; d=DList->firstNode; OutputList=Null;
    stack->setempty();
3   while((AList->isempty() == false) || (DList->isempty() ==
4   false) || (stack->isempty() == false)) {
5      if     ((a.startpos     >     stack->top.endpos)     &&
    (d.startpos>stack->top.endpos))
    /* time to pop the top element in the stack*/
6         stack->pop();
7      else if (a.startpos < d.startpos) {
8         stack->push(a);
9         a=a->nextNode;
10     }
11     else {
    /* append ancestors to OutputList and make stack empty*/
12     for(a1=stack->bottom; a1!=NULL; a1=a1->up)
13        append a1 to OutputList
14        stack->reset();
15        d=d->nextNode;
16     }
17  }
18  Return OutputList;
```

그림 3 NList-SSJoin-Anc 알고리즘

와 같은 연결리스트에 소요되는 계산 비용을 줄일 수 있다. Self와 Inherit 리스트의 크기는 이어지는 노드가 많을수록 더욱 커진다. 이것은 XML문서가 크거나 태그가 많을 경우, 더욱 뛰어난 효율성을 나타낼 수 있다는 것을 알 수 있다. 그림 3은 NStack-SSJoin-Anc 알고리즘을 나타낸다. 알고리즘의 자세한 처리과정에 대한 예는 구조적 세미조인[1]을 참고하면 된다.

3. 구조적 세미조인을 이용한 다중경로패턴 처리

이번 장에서는 구조적 세미조인을 이용한 다중경로패턴처리 알고리즘(PathJoin algorithm)을 제시한다. 이 알고리즘은 본 논문에서 소개한 구조적 세미조인(Structural-SemiJoin)과 스택기반의 구조적 세미조인 알고리즘(Naïve-SSJoin)을 전 방향(Forward) 또는 역방향(Backward)으로 적용하여 처리하는 알고리즘이다. 그림 4와 그림 5는 전 방향 경로패턴처리 알고리즘과 역방향 경로패턴처리 알고리즘을 보여준다.

```
1   Algorithm PathJoin_Foward (NumofTags, Tagnames)
2   AList <- Node List of Tagnames[0];
3   DList <- Node List of Tagnames[1];
4   for(i=1;i< NumofTags;i++) {
5      OutputList <- Sem_Join_Desc(AList, DList);
6      AList <- OutputList;
7      DList <- Node List of Tagnames[i+1];
8   }
9   Return OutputList;
```

그림 4 전방향 경로패턴처리 알고리즘

```
1   Algorithm PathJoin_Backward(NumofTags, Tagnames)
2   AList <- Node List of Tagnames[NumofTags-2];
3   DList <- Node List of Tagnames[NumofTags-1];
4   for(i=NumofTags-1;i>0;i--)  {
5      OutputList <- Sem_Join_Anc(AList, DList);
6      DList <- OutputList;
7      AList <- Node List of Tagnames[i-2];
8   }
9   Return OutputList;
```

그림 5 역방향 경로패턴처리 알고리즘

4. 실험결과

본 논문에서 제시한 구조적 세미조인을 사용할 경우와 기존의 스택기반의 구조적 세미조인을 사용했을 때 경로처리 비용을 비교하였다. 실험에서는 200M byte 크기의 XML 문서를 사용한다. 이 문서의 DTD는 그림 6에 나타나 있다. Department 엘리먼트는 name, email, manager, employee 그리고 department가 한 번 더 들어가 있다. Manager와 employee는 name과 employee

```
<?xml version="1.0" encoding="UTF-8"?>
<!ELEMENT department (name+, email?, manager*, employee+, department*)>
<!ELEMENT manager (name+,email?)>
<!ELEMENT employee (name+,email?)>
<!ELEMENT name (#PCDATA)>
<!ELEMENT email (#PCDATA)>
```

그림 6 실험에 쓰일 DTD

표 1 Database 상태

Tag Name	Number of Tags
Department	397,947
Manager	597,302
Employee	796,671
Name	3,383,862
Email	895,957

표 2 경로처리 실험에 대한 XPath질의

No	Query	Path	Return Type	# of Results
1	department//department //name//name	4	Desc	94,836
2	department[department [manager[name]]]	4	Anc	17
3	department//department //manager	3	Desc	398,768
4	department[department [manager]]	3	Anc	17
5	department//employee //email	3	Desc	897,301
6	department [employee[email]]	3	Anc	286,311
7	department//department //department//email	4	Desc	895,954
8	department[department [department[email]]]	4	Anc	8

가 서브엘리먼트로 들어가 있다. 각각의 엘리먼트 수는 표 1에 나타나 있다. 표 2는 실험에 쓰인 XPath질의를 나타낸다. 리턴 타입은 역 방향으로 처리하는 경우와 전방향으로 처리하는 경우를 나타내며, Path는 경로의 개수, 마지막은 질의 결과의 개수를 나타낸다.

전방향 경로패턴처리 실험에서는 스택에 기반한 알고리즘인 Naïve-SSJoin-Desc와 개선된 알고리즘인 NStack-SSJoin-Desc을 비교하였고, 역방향 경로패턴처리 실험에서는 스택에 기반한 알고리즘인 Naïve-SSJoin-Anc와 개선된 알고리즘인 NList-SSJoin-Anc을 비교하였다.

그림 7과 8은 각각 전방향 및 역방향 경로처리 실험 결과를 나타낸다. 그래프의 X축은 질의의 번호를 나타내고, Y축은 알고리즘의 반응 시간을 나타낸다. 그림 7을 보면, 전방향 처리시, 기존의 구조적 조인 방식에 비해 개선된 구조적 세미조인 방식을 사용할 때, 최대 20

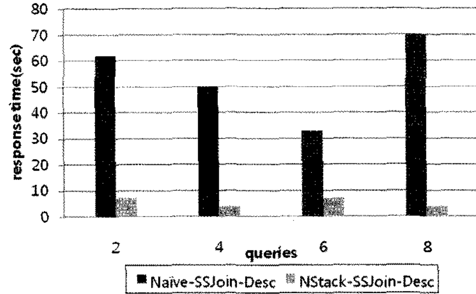

그림 7 전방향 경로처리 성능 비교

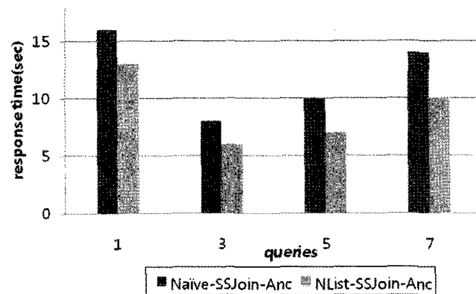

그림 8 역방향 경로처리 성능 비교

배의 빠른 성능을 나타내었다. 또한 그림 8을 보면, 역방향 처리시에도 개선된 방식이 2배 가까운 빠른 성능을 나타냄을 알 수 있다. 그림 7과 8을 비교해 보면 기존의 방식은 전방향이 역방향에 비해서 매우 느린 성능을 나타냄을 알 수 있는데, 이는 전방향 방식이 중간 결과를 더 많이 발생시키기 때문인 것으로 분석된다.

6. 결 론

본 논문에서는, 기존의 구조적 조인을 개선한 구조적 세미조인 알고리즘과 이를 이용한 XML 경로처리 알고리즘을 제시하였다. 실제 데이터를 사용한 실험 결과에서 전방향 혹은 역방향으로 탐색하는 경로 처리시 개선된 구조적 세미조인 알고리즘을 사용할 때, 기존의 알고리즘을 사용할 때보다 더 좋은 성능을 나타냄을 알 수 있었다. 특히, 역방향 경로 처리의 경우 최대 20배 이상의 성능 차이를 보여주었다.

참 고 문 헌

[1] S. Son, H. Shin and Z. Xu, "Structural Semi-Join : A light-weight structural operator for efficient XML query pattern matching," In *Proc. of 2007 International Database Engineering & Application Symposium*, pp.233-240, September 2007.

[2] D. Srivastava, S. A.-Khalifa, H. V. Jagadish, N. Koudas, J. M. Patel, and Y. Wu, "Structural joins:

A primitive for efficient XML query pattern matching," In *Proc. of 2002 International Conference on Data Engineering*, pp.141-152, February 2002.

[3] S.-Y. Chien, Z. Vagena, D. Zhang, V. J. Tsotras, and C. Zaniolo, "Efficient structural joins on indexed XML documents," In *Proc. of 2002 International Conference on Very Large Data Bases*, pp.263-274, August 2002.

[4] H. Li, M.-L. Lee, W. Hsu, and C. Chen, "An evaluation of XML indexes for structural join," *ACM SIGMOD Record*, 33(3), pp.28-33, April 2004.

[5] K.-L. Wu, S.-K. Chen, and P. S. Yu, "Efficient structural joins with on-the-fly-indexing," In *Proc. of 2005 International Conference on World Wide Web*, pp.1028-1029, May 2005.

[6] C. Luo, Z. Jiang, W.-C. Hou, F. Yan, and C.-F. Wang, "Estimating XML structural join size quickly and economically," In *Proc. of 2006 International Conference on Data Engineering*, pp.62-62, April 2006.

[7] C. Mathis and T. Harder, "Hash-based structural join algorithms," In *Proc. of 2006 International Conference on Extending Database Technology*, pp.136-149, March 2006.

[8] C. Mathis, T. Harder, and M. P. Haustein, "Locking-aware structural join operators for XML query processing," In *Proc. of 2006 International Conference on Special Interest Group on Management Of Data*, pp.467-478, June 2006.

[9] Y. Wu, J. M. Patel, and H. V. Jagadish. "Structural join order selection for XML query optimization," In *Proc. of 2003 International Conference on Data Engineering*, pp.443-454, March 2003.

[10] P. Mandawat and V. J. Tsotras, "Indexing schemes for efficient aggregate computation over structural joins," In *Proc. of 2005 International Workshop on Web & Data Bases*, pp.55-60, June 2005.

[11] K. Liu, F. H. Lochovsky, "Efficient computation of aggregate structural joins," In *Proc. of 2003 International Conference on Web Information Systems Engineering*, pp.21-30, December 2003.

[12] N. Bruno, N. Koudas, and D. Srivastava, "Holistic twig joins: optimal XML pattern matching," In *Proc. of 2002 International Conference on Special Interest Group on Management Of Data*, pp. 310-321, June 2002.

[13] C. Zhang, J. F. Naughton, Q. Luo, and D. J. DeWitt, and G. M. Lohman, "On supporting containment queries in relational database management systems," In *Proc. of 2001 International Conference on Special Interest Group on Management Of Data*, pp.425-436, May 2001.

[14] Q. Li and B. Moon, "Indexing and querying XML data for regular path expressions," In *Proc. of 2001 International Conference on Very Large Data Bases conference*, pp.361-370, September 2001.