XML 문서저장 시스템을 위한 DOM 인터페이스의 설계 및 구현

김성욱, 정호영, 김천식, 손기락

한국외국어대학교 컴퓨터공학과

Design and Implementation of DOM Interface for an XML Document Storage System

SeongWook Kim, HoYoung Jeong, ChunSik Kim, Kirack Sohn

Dept. of Computer Science & Engineering, Hankuk University of Foreign Studies

요약

구조적 정보의 표현방법으로 제시된 XML 문서를 가공하고 저장하는 방법에 대한 연구가 활발하게 진행되고 있다. XML 을 가공하는 형태로는 몇 가지가 제시되어 일반화 되어있으나 계층적 특성을 가진 XML 문서를 관계형 데이터베이스에 효율적으로 저장하기는 어렵다. 본 연구에서는 XML 문서의 계층 구조를 DFS Numbering 으로 저장하고 데이터베이스에 저장된 문서에 대한 DOM 인터페이스를 효과적으로 제공하는 시스템의 설계 및 구현이다. 문서의 변경 내용을 저장할 때 SQL Query 횟수를 최소화하는 효율적인 기법을 제시한다.

1. 서론

최근 인터넷이 보편화되고 그에 따른 정보의 양도 급증함으로 인해 좀 더 효율적인 정보의 전달 방법이 필요하게 되었다. 이에 맞추어 W3C(World Wide Web Consortium)에서는 구조화된 정보의 표현 방법으로 XML(Extensible Markup Language)[3]을 발표하였고 이를 Access 할 수 있는 interface 로 DOM(Document Object Model)[4]을 제시하였다. DOM 은 XML 문서를 Tree 형태로 나타낸 것으로써 XML 의 큰 특징중의 하나인 계층적 정보를 가장 잘 표현하는 interface 라 할 수 있다.

근래에 이러한 XML 문서의 효과적인 저장과 관리를 위한 연구가 활발히 진행되고 있는데 본 논문에서 제시하고자 하는 것은 XML 문서를 DOM 형태로 표현하고 이를 데이터베이스화하는 것으로, 효율적인 정보의 저장과 편리한 정보의 가공을 가능하게 한다. XML 문서를 DOM Interface 로 변환하여 변환된 Tree 에 DFS Numbering 을 통한 저장한다. XML 문서는 문서 전체로 Check-in/Check-out 되고, Check-in 시 변경된 내용은 데이터베이스에 반영된다. 이 때 저장 시 필요한 SQL 질의의 수를 최소화하는 방안을 제시한다. XML 문서 저장을 위한 데이터베이스 스키마는 [1][2]를 참조한다.

2. 효율적인 DB Access 를 위한 자료구조

본 절에서는 데이터베이스에서 문서를 읽어와서 가공하고 다시 Table 에 저장하는 효율적인 방법을 제시한다.

Node Table 에 나와있는 start 와 end 값은 Tree 를 DFS Search 하면서 Numbering 한 값으로 start 는 방문할 때 값을 나타내며, end 는 방문이 끝나고 다음 Node 로 이동할 때 값을 나타낸다. 이는 Node Table 에서 어느 특정한 docId 로 Node Table 을 읽어올 때 start 값으로 정렬하면 Tree 를 Preorder 로 Search 한 결과와 같은 형태로 된다는 것을 의미한다. 따라서 Table 에 있는 문서로부터 DOM Tree 를 구성할 때 start 값과 end 값을 참고한다면 원래의 형태를 유지할 수가 있다. 이를 위해서는 저장할 때에도 Tree 를 DFS Search 하여 순서대로 저장할 필요가 있는데 단순히 DFS Search 만으로 문서를 저장한다면 해당 Document 의 Node Table 정보를 모두 삭제하고 처음부터 순서대로 다시 Insert 하는 방법밖에 없을 것이다. 이는 대용량의 문서에 대해서는 엄청난 Disk Access 횟수와 Query 의 실행 횟수로 인해 많은 비용이 요구되어 비효율적이다.

각 Node 의 dbInfo 는 원래 Table 로부터 읽어온 start, end 값, 그리고 이후에 수정된 start, end 값을 offset 개념으로 유지하고 있는데 그 구조는 <표 1>과 같다.

<표 1> dbInfo 의 구조

Name	Type	Comment
start	int	Table 에 저장된 start 값
end	int	Table 에 저장된 end 값
moreStart	int	수정된 start 값과 Table 에 저장된 start 값과의 차
moreEnd	int	수정된 end 값과 Table 에 저장된 end 값과의 차
deleted	boolean	Node 의 삭제여부
modified	boolean	nodeValue 의 수정여부
node	Node	dbInfo 의 소유 Node

dbInfo 는 각 Node 마다 할당되어 있는데 XML Manager 객체가 Tree 전체의 dbInfo 를 Vector 형태로 표현된 dbInfoTable 을 가지고 있으며 이를 통해 데이터베이스에 저장하고 읽을 수 있다. XML Manager 는 DOM Tree 의 Root 에 해당되는 Document Node 에 할당되어 관리된다.

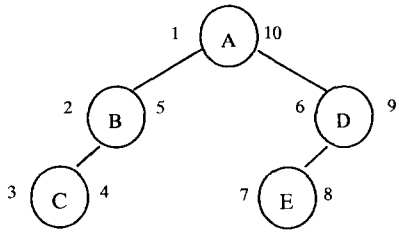
<그림 1>과 같은 DOM Tree 가 데이터베이스로부터 Loading 되었을 경우를 예를 들어 설명한다.

2.1 새로운 Node F 의 삽입

기존의 DOM Tree 에 새로운 Node F 가 삽입된 DOM Tree 와 dbInfoTable 의 형태는 <그림 2>와 같다.

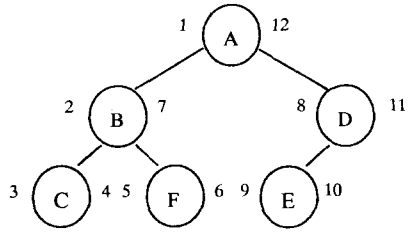
새로운 Node 가 삽입되었을 경우에는 기존의 dbInfo 의

Node	Start	End	MS	ME	Deleted	Modified
A	1	10	0	0	False	False
B	2	5	0	0	False	False
C	3	4	0	0	False	False
D	6	9	0	0	False	False
E	7	8	0	0	False	False



<그림 1> 데이터베이스로부터 읽어온 초기상태

Node	Start	End	MS	ME	Deleted	Modified
A	1	10	0	+2	False	False
B	2	5	0	+2	False	False
C	3	4	0	0	False	False
D	6	9	+2	+2	False	False
E	7	8	+2	+2	False	False
F	0	0	+5	+6	False	False



<그림 2> 새로운 Node F 삽입후의 상태

moreStart, moreEnd 에 삽입하기 전의 start, end 값에 대한 삽입 후의 start, end 값의 변화량을 저장하고 새로운 Node 의 dbInfo 를 dbInfoTable 의 끝에 삽입한 후에 start, end 값을 0, moreStart, moreEnd 에 현재 start, end 값을 저장한다.

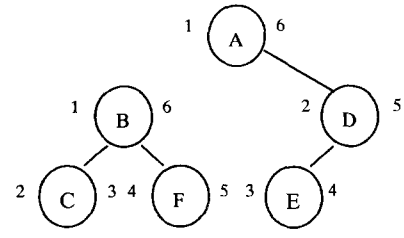
2.2 Node B 의 삭제

기존에 존재하던 Node B 가 삭제될 경우 Node B 를 Root 로 하는 모든 Node 에 대해 deleted 값을 True 로 설정해 준다. 단, 새로 삽입된 Node F 에 대해서는 dbInfo 를 dbInfoTable 로부터 제거한다. 그리고 Node B 를 시작으로 DFS Search 하여 Numbering 을 다시 한 후 삭제전의 start, end 값에 대한 삭제후의 start, end 값의 변화량을 moreStart, moreEnd 에 저장한다. 이는 실질적으로 Node A 의 Start 값 만큼을 Node B 와 그 Child 들의 moreStart, moreEnd 에서 빼 주는 것이다. 이 Node 들이 제거 된 후에는 나머지 DOM Tree 의 start, end 값을 조정하여 그 변화량 만큼을 moreStart, more End 에 저장한다. <그림 3> 이 결과를 나타내고 있다.

2.3 삭제된 Node B 를 Node D 의 뒤에 삽입하였을 경우

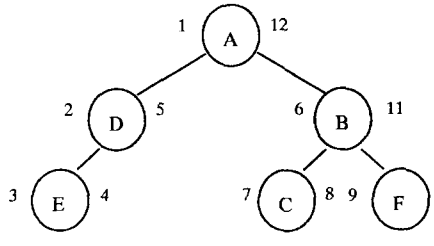
삭제된 Node B 를 Node D 의 뒤로 삽입할 경우에는 Node B 를 Root 로 하는 Tree 에 Node D 의 end 값 만큼을 moreStart, moreEnd 에 더해줘서 삽입된 후에 DFS Numbering 이 유지될 수 있도록 하고, Node B 가 삽입됨으로 인해 변화되는 Node 들의 start, end 값을 조정한다. 이때 2.2 에서 제거된 새로운 Node F 의 dbInfo 는 다시 dbInfoTable 의 끝에 삽입되며, Node B 와 C 의 deleted 값은 False 가 된다. 이러한 작업후의 형태는 <그림 4>와 같다.

Node	Start	End	MS	ME	Deleted	Modified
A	1	10	0	-4	False	False
B	2	5	-1	+1	True	False
C	3	4	-1	-1	True	False
D	6	9	-4	-4	False	False
E	7	8	-4	-4	False	False



<그림 3> Node B 를 삭제한 후의 상태

Node	Start	End	MS	ME	Deleted	Modified
A	1	10	0	+2	False	False
B	2	5	+4	+6	False	False
C	3	4	+4	+4	False	False
D	6	9	-4	-4	False	False
E	7	8	-4	-4	False	False
F	0	0	9	10	False	False



<그림 4> 삭제된 Node B 를 Node D 의 뒤에 삽입한 후의 상태

2.4 Node E, F 의 값이 수정되었을 경우

새로 삽입된 Node 나 기존에 존재하던 Node 나 상관없이 nodeValue 가 수정되었을 경우에는 modified 값을 True 설정하여 값이 수정되었음을 명시한다.

3. DOM Tree 의 저장

2 절에서 제시한 예에서 보듯이 데이터베이스로부터 읽어온 start, end 값은 전혀 변화되지 않고 Tree 구조나 Node 의 값이 변경되었을 경우에 그 변화의 정도, 여부를 기록하고 있다. 앞에서도 언급했지만 매번 이런 변화의 정도, 여부를 기록하는 목적은 DOM Tree 를 데이터베이스에 저장할 때의 비용을 줄이고자 함이다. 이를 구현한 알고리즘은 다음과 같다.

```
S1.  set currentState := INSERT_MODE;
     set updateStart := 0, updateEnd := 0;
     set moreStart := 0, moreEnd := 0;
     set I := 0
S2.  previousState := currentState;
S3.  if dbInfoTable[I].deleted = False then go to S5;
     // 삭제된 노드가 아닌 경우
S4.  if previousState = UPDATE_MODE then begin
       update(updateStart, updateEnd, moreStart, moreEnd);
       updateStart := 0;
     end;
     set currentState := DELETE_MODE;
     set updateEnd := dbInfoTable[I].end;
     if updateStart = 0 then updateStart := dbInfoTable[I].start;
     go to S10;
S5.  if dbInfoTable[I].modified = False then go to S7;
     // nodeValue 가 수정되지 않은 경우
S6.  updateValue();
     go to S10;
S7.  if dbInfoTable[I].start <> 0 then go to S9;
     // 기존의 DB 정보인 경우
S8.  insertNewNode();
     go to S10;
S9.  if dbInfoTable[I].moreStart <> 0 or // Tree 의 구조변경
     dbInfoTable[I].moreEnd <> 0 then begin
     if previousState = DELETE_MODE then begin
       delete(updateStart, updateEnd);
       updateStart := 0;
     end
     else if previouseState = UPDATE_MODE then begin
       // 이전과 같은 변화량을 가지는지를 검사
       if moreStart <> dbInfoTable[I].moreStart or
          moreEnd <> dbInfoTable[I].moreEnd then begin
          update(updateStart, updateEnd, moreStart, moreEnd);
          updateStart := 0;
       end;
     end;
     currentState := UPDATE_MODE;
     moreStart := dbInfoTable[I].moreStart;
     moreEnd := dbInfoTable[I].moreEnd;
     updateEnd := dbInfoTable[I].end;
     if updateStart = 0 then updateStart := dbInfoTable[I].start;
S10.  I := I + 1;
S11.  if I < size of dbInfoTable then go to S2;
S12.  if currentState = DELETE_MODE then
          delete(updateStart, updateEnd);
      else
          update(updateStart, updateEnd, moreStart, moreEnd);
```

위의 알고리즘은 dbInfoTable 을 처음부터 순차적으로 읽으면서 데이터베이스에 저장하는 방법을 사용하고 있는데, nodeValue 의 update 나 새로운 Node 의 삽입의 경우에는 평이한 방법으로 Node 하나씩 Query 문을 작성하여 실행하고 있다. 그러나 삭제된 기존의 Node 나 구조가 변경된 Node 에 대해서는 연속적으로 나타나는 속성을 고려하여 한꺼번에 Query 를 실행한다는 것이다.

위의 알고리즘대로 앞의 예제들을 데이터베이스에 저장한다면 그 결과는 <표 4>와 같이 나타난다.

<표 4> 예제의 저장 시 Query 실행횟수

예제	Query 실행횟수
그림 1	0
그림 2	Update * 2 ({A,B},{D,E}), Insert * 1 ({F})
그림 3	Update * 2 ({A},{D,E}), Delete * 1 ({B,C})
그림 4	Update * 4 ({A},{B},{C},{D,E}), Insert * 1 ({F})

4. 결론

본 논문에서는 DFS Numbering 을 이용하여 DOM Tree 를 효율적으로 데이터베이스에 저장하는 방법을 제시하였다. 실질적으로 큰 XML 문서에 대해 Query 실행 횟수나 속도면에서 좋은 성능을 보였다.

제시된 방법은 DOM Tree 에서 어떤 형태의 구조변경이 있느냐에 따라 Query 실행횟수와 속도가 차이가 있어 정확한 실험 결과를 제시하기가 힘들다. 하지만, 저장된 Node 정보를 모두 지우고 다시 삽입하는 방법에 비해 월등한 비용의 절약이 있었다.

5. 참고문헌

[1] 이용석, 손기락, "XML 문서 저장 시스템 설계 및 구현", 한국정보과학회 추계학술발표 논문집, 1998.

[2] 허명식, 손기락, "XQL 을 지원하는 XML 문서 저장 시스템", 한국정보과학회 추계학술발표 논문집, 1999.

[3] Extensible Markup Language (XML), Version 1.0. W3C Recommendation. Available at http://www.w3.org/TR/1998/REC-xml-19980210.

[4] Document Object Model (DOM), Level 2 Specification. W3C Candidate Recommendation. Available at http://www.w3.org/TR/1999/CR-DOM-Level-2-19991210.