유효한 XML 환경에서의 효율적인 갱신 충돌 탐지 기법

변창우○, 윤일국, 박 석
서강대학교 컴퓨터공학과
{chang○, bingo619, spark}@sogang.ac.kr

An Efficient Detection of Conflicting Updating in valid XMLs

Changwoo Byun○, Ilkook Yun, Seog Park
Department Computer Science and Engineering , Sogang University

요 약

XML 전용 데이터베이스 시스템의 등장 및 갱신 연산 지원되면서 갱신 연산의 유효성 검사 및 효율적인 갱신 연산의 충돌 감지 기법의 필요성이 대두되고 있다. 이러한 필요성은 잘 정형화된 XML 문서와는 달리 스키마의 제약사항을 준수해야 하는 유효한 XML 문서 환경에서 절실히 요구된다. 특히, 효율적인 갱신 연산의 충돌 탐지 기법은 질의 컴파일러의 질의 최적화 및 트랜잭션 관리의 높은 병행수행 목적을 달성하는데 필수적인 요소이다. 본 논문은 판독-갱신 및 갱신-갱신 연산 사이의 충돌을 정의하고, 유효한 XML 환경에서 효율적으로 충돌을 감지하는 기법을 제안한다.

1. 서론

최근에 잘 정형화된 (well-formed) XML에 대한 갱신 연산 및 효율적인 처리 기술에 대한 연구가 진행되었다 [1,2]. 또한, 유효한 (valid) XML에서 갱신 연산 후 변경된 문서와 스키마 사이의 검증 작업이 필요하다. 메모리 공간 및 검증 시간의 최적화를 위해 갱신 연산에 의해 변경된 문서 일부분에 대한 점진적 유효성 검증에 대한 많은 연구가 진행되었다 [3-7].

이와 같은 갱신 연산의 지원에 의해 추가적으로 진행되고 있는 연구 분야는 연산들 간의 병행수행을 지원하는 트랜잭션 관리 기술이고, 트랜잭션 관리 기술에 기본적으로 해결해야 하는 부분이 연산 간의 충돌을 탐지하는 기법이다 [10-12]. M. Raghavachari과 O. Shmueli은 최근의 연구에서 XPath$^{/,\ //,\ *,\ []}$ 경로 표현식 기반의 연산 간의 충돌 탐지는 NP-complete임을 보이고, 프레디키트가 없는 XPath$^{/,\ //,\ *}$ 경로 표현식 기반의 판독-삽입, 판독-삭제 연산 간의 충돌을 탐지하는 Polynomial 알고리즘을 제안하였다 [13]. 그러나, 이 연구는 잘 정형화된 XML 환경을 가정으로 하고 있다.

본 논문에서는 유효한 XML 환경에서 적절한 스키마 정보를 이용하여 연산들 간의 효율적인 충돌 탐지 기법을 제안한다. 제안하는 방법은 복잡도를 심각하게 고려하지 않아도 되는 장점과 함께 다음과 같은 기여도 갖고 있다:

- 판독-갱신 및 갱신-갱신 연산 사이의 충돌에 대한 개념 정의
- 유효한 XML 환경에서의 효율적이고 효과적인 연산 간의 충돌 탐지 기법 제안
- 제안하는 충돌 탐지 기법은 유효한 XML 데이터베이스와 독립성을 추구할 수 있어 어떠한 유효한 XML 데이터베이스 관리 시스템에 탑재 가능

본 논문의 구성은 다음과 같다: 2장에서는 갱신 연산에 대한 간략한 의미와 관련 연구를 소개한다. 3장에서 유효한 XML 환경에서 새로운 스키마 정보 및 충돌 개념을 정의하고 제안된 스키마 정보를 이용한 효율적인 충돌 탐지 기법을 제안한다. 4장에서는 갱신-갱신 연산의 충돌 중 삭제-갱신 연산의 충돌을 완화할 수 있는 경우에 대해 토론한다. 최종적으로 5장에서 결론을 맺고 추후 연구를 소개한다.

2. 관련 연구

2. 1 XML

일반적으로 XML은 잘 정형화된 XML과 유효한 XML 두 가지 유형으로 나뉜다. 유효한 XML은 잘 정형화된 XML이면서 스키마 (DTD 혹은 XML Schema)의 제약사항을 준수하는 XML 문서를 말한다. DTD는 엘리먼트 타입과 각각의 엘리먼트의 특성을 기술한다. 만약 XML 문서가 DTD를 가지고 있고 그 DTD의 제약사항을 만족하고 있다는 그 문서를 유효하다고 한다 [8].

엘리먼트 유형은 어떤 자식 엘리먼트를 가지고, 그들 간의 순서 (예, sequence (‘,’로 표현) or choice (‘|’로 표현))는 어떻게 되는지 기술하고 인스턴스 상의 발생 횟수 (예, ? (없거나 한번), * (횟수에 제안 없음), + (한번 이상))를 제약한다. 추가로 엘리먼트 간의 순서 및 발생 횟수가 임의의 자식 엘리먼트들 간에 그룹을 형성할 수 있게 하고 있다. 지면상 보다 자세한 설명은 생략한다. 그림 1 은 본 연구에서 이용하는 DTD 로 최근 신문사나 잡지사 혹은 블로그와 같은 웹 사이트에서 컨텐츠

이 논문은 2006년 정부(교육인적자원부)의 재원으로 한국학술진흥재단의 지원을 받아 수행된 연구임 (KRF-2006-102483-D00848)

정보가 요약된 RSS (Real Simple Syndicate)라는 XML 포맷에 대한 DTD 이다.

```
<!ELEMENT rss (channel)*>
<!ELEMENT channel (title, link, description, lastmodified,(author | editor)?, hit, rank?, item*)>
<!ELEMENT item (title, link, description, author, pubdate)>
<!ELEMENT title #PCDATA>
<!ELEMENT link #PCDATA>
<!ELEMENT description #PCDATA>
<!ELEMENT lastmodified #PCDATA>
<!ELEMENT author #PCDATA>
<!ELEMENT editor #PCDATA>
<!ELEMENT hit #PCDATA>
<!ELEMENT rank #PCDATA>
<!ELEMENT pubdate #PCDATA>
```

그림 1 DTD 예제

[**가정 1**] 본 논문은 DTD 환경의 유효한 XML 환경을 가정한다.

2.2 XML 갱신 연산

최근 XML 문서에 대한 갱신 연산에 대한 연구가 진행되고 있다. Igor Tatarinov는 INSERT, DELETE, REPLACE 그리고 RENAME 연산을 제안했으며 [1]. M. Rys는 추가적으로 MOVE 연산을 제안하였다 [2].

본 논문은 INSERT, DELETE, 그리고 REPLACE 연산을 가정하며 이들 연산들을 통틀어 UPDATE 연산이라 한다. 비록 REPLACE 연산은 순서화된 XML 모델에서 INSERTBEFORE 연산 후에 DELETE 연산을 한 결과와 같고, 순서화되지 않은 XML 모델에서는 INSERT 연산 후에 DELETE 연산을 한 결과와 같지만, 편이성에 의해 필요한 연산이기 때문에 개별적인 연산으로 취급한다.

INSERT, DELETE, 그리고 REPLACE 연산에 대한 타입을 Igor Tatarinov가 제안한 것을 그대로 인용한다 [1]:

- INSERT(*NodeSet, child*): INSERT는 *NodeSet*의 자식으로 새로운 데이터 (*child*)를 삽입한다. 삽입되는 자식 노드는 엘리먼트와 속성 (attribute)이 될 수 있다.
- DELETE(*NodeSet*): DELETE는 *NodeSet*의 모든 자식 노드와 *Nodeset* 자체를 삭제한다.
- REPLACE (*NodeSet, old_content, new_content*): REPLACE는 *NodeSet*의 자식 노드인 *old_content*를 새로운 노드인 new_content로 교체한다.

표 1은 확장된 갱신 XQuery 표현과 본 논문에서 이용하는 연산과의 관계를 보여주고 있다.

표 1 확장된 갱신 XQuery와 관련된 연산

확장된 갱신 XQuery 표현식	관련된 연산
FOR $j *IN* /rss/channel/author *RETURN* $j	READ($j)
LET $bb := /rss/channel $cc IN $bb/item[2]	
UPDATE $bb { *REPLACE $bb/hit WITH* *<hit> 317046 </hit>* *DELETE $cc* *INSERT* <item>…</item> }	 REPLACE($bb/hit, <hit> 317046 </hit>) DELETE ($cc) INSERT($bb, <item>… </item>)

2.3 충돌 탐지에 대한 기존 연구

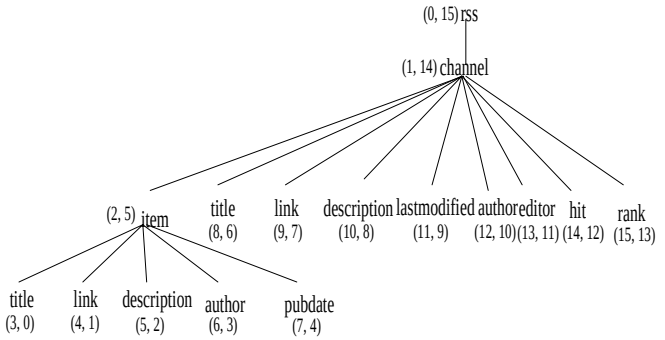
일반적으로 XML 환경에서 연산 간의 충돌 탐지에 대한 연구는 트랜잭션 관리 기법에서 소개되고 있다. S. Helmer 는 XML 환경의 트랜잭션 관리 기법으로 네 개의 로킹 기반 프로토콜 (Doc2PL, Node2PL, NO2PL, 그리고 OO2PL)을 제안하고 있다 [10]. T. Grabs 는 경로 기반의 로킹 기법인 DGLOCK을 제안하고 있다 [11]. S. Dekeyser는 경로 로킹 프로토콜인 경로 로크 전파 (path lock propagation) 및 경로 로크 해갈 (path lock satisfiability) 프로토콜을 제안하고 있다. 이와 같이 언급한 연구들은 로크 호환 행렬 (lock compatibility matrix)을 기반으로 충돌을 탐지한다. 즉, 실제 XML 인스턴스에 로크를 놓고, 트리 구조를 경유하여 들어오는 로크와 로크 호환 행렬을 살펴보면서 충돌을 탐지한다. 따라서, XML 인스턴스에 많은 로크가 존재하게 되고, 트리 구조의 전이를 통해 실제 노드에서 충돌을 탐지하기 때문에 탐지 시간도 늘어나는 문제점을 가지고 있다.

M. Raghavachari과 O. Shmueli는 READ-INSERT와 READ-DELETE에 대해 노드 충돌과 트리 충돌이라는 한 새로운 충돌 개념을 소개하고 있다 [13]. 또한, 그들은 XPath$^{/,\ //,\ *,\ []}$ 경로 표현식 기반의 연산 간의 충돌 탐지는 NP-complete임을 보이고 있으며, 프레디키트가 없는 XPath$^{/,\ //,\ *}$ 경로 표현식 기반의 연산 간의 충돌을 탐지하는 Polynomial 알고리즘을 제안하였다. 그러나, 이 연구는 잘 정형화된 XML 환경을 가정으로 하고 있다.

3. 효율적인 충돌 탐지 알고리즘

3.1 스키마 정보

그림 2(a)는 그림 1에서 제시한 DTD를 순차적으로 선회를 한 PRE(order)와 POST(order) 값이 할당된 노드들에 대한 DTD 트리를 보여주고 있다.



(a)

Tag-Name	Pre	size	level	post
rss	0	15	0	15
channel	1	14	1	14
item	2	5	2	5
title	3	0	3	0
link	4	0	3	1
description	5	0	3	2
author	6	0	3	3
pubdate	7	0	3	4

Tag-Name	Pre	size	level	post
title	8	0	2	6
link	9	0	2	7
description	10	0	2	8
lastmodified	11	0	2	9
author	12	0	2	10
editor	13	0	2	11
hit	14	0	2	12
rank	15	0	2	13

(b)

그림 2: (a) 그림 1의 DTD 트리 구조, (b) DTD 트리 구조에 의한 스키마 정보

그림 2(b)는 그림 2(a)에 대한 스키마 정보를 보여주고 있다. 이를 PRE/POST 구조 (PPS)라 명명한다. LEVEL를 DTD 트리 상에서의 루트를 기반으로 한 레벨을 의미하며, SIZE는 임의의 노드의 하위 트리의 모든 노드들의 개수를 말한다. 이런 PRE/SIZE/LEVEL 인코딩은 POST = PRE + SIZE – LEVEL 의 수식에 의해 한번의 선회로 구할 수 있다 [14]. 3.3절의 충돌을 탐지하는 기법에 이 정보를 이용하게 된다.

3.2 Conflicting XML Updates

이번 절에서는 XML 데이터 환경에서 READ-UPDATE와 UPDATE-UPDATE 연산에서의 충돌을 정의한다.

문서의 루트를 기반으로 어떤 노드까지의 XPath 경로 표현식 기반의 READ 연산은 그 노드의 하부 트리에 대한 판독 연산이다 [15]. 같은 의미의 XPath 경로 표현식을 사용하는 UPDATE 연산은 그 노드의 하부 트리의 일부분에 대해 갱신을 수행하는 연산이다.

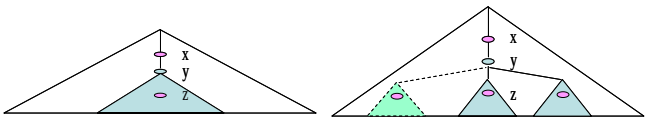
자세하게 연산 간의 충돌을 정의하기에 앞서 READ 연산와 UPDATE 연산은 *유효한 연산* (valid operation)임을 가정한다. *유효한 연산* 이라 함은 XPath 경로 표현식에 있는 구조 정보는 DTD의 구조 정보에 어긋나지 않는 연산을 말한다. 예를 들어, 그림 1의 DTD를 기준으로 READ(rss/channel/title) 연산은 *유효한 연산*이지만, READ(rss/channel/title/author)은 그렇지 않다. 왜냐하면 구조 정보에 의해 *author* 노드는 *title* 노드의 자식으로 올 수 없기 때문이다.

XML 문서는 임의의 알파벳 ∑에 속한 심볼로 레이블화된 노드들로 이루어진 트리라 하고 알파벳 ∑ 상의 모든 트리의 집합을 T_Σ라 표시했을 때, READ 연산 R_i 와 UPDATE 연산 U_j 의 충돌을 다음과 같이 정의된다.

정의 1 [**READ-UPDATE 충돌**] 임의의 $t \in T_\Sigma$에 대해 $R_i(U_j(t)) \neq R_i(t)$이면 R_i 와 U_j 는 충돌이다.

$R_i(U_j(t)) \neq R_i(t)$의 의미는 READ 연산의 XPath 경로 표현식에 의한 하부 트리 영역이 UPDATE 연산의 XPath 경로 표현식에 의한 하부 트리 영역을 포함하고 있으면, 결국 READ 연산은 UPDATE 연산의 결과를 판독하게 된다는 의미이다.

그림 3 (a)에서 보듯이, READ 연산 R_i의 하부 트리의 루트를 x, DELETE 연산 U_j의 하부 트리의 루트를 y라 하면, x 노드는 y 노드의 조상 (ANCESTOR) 노드이고, $R_i(U_j(t))$는 트리 t 상에서의 회색 부분을 배제한 부분을 판독한다. 그러나, $R_i(t)$는 회색 부분을 판독하게 되어 그 결과가 다르다 ($R_i(U_j(t)) \neq R_i(t)$). 만약 READ 연산의 하부 트리의 루트 (z)가 후손 (DESCENDANT) 노드인 경우에도 역시 READ 연산의 결과는 다르다 ($R_i(U_j(t)) \neq R_i(t)$). 그림 3(b)에서는 READ 연산과 INSERT (혹은 REPLACE)에서의 충돌 상황을 보여주고 있다. 유사한 방법으로 해석하면, 갱신 연산 전후의 READ 연산의 결과는 다르다 ($R_i(U_j(t)) \neq R_i(t)$).



(a) READ-DELETE conflict (b) READ-INSERT (or REPLACE) conflict

그림 3 READ-UPDATE 충돌 예

다음으로 UPDATE 연산 U_i와 U_j에서의 충돌을 정의한다.

정의 2 [**UPDATE-UPDATE 충돌**] 임의의 $t \in T_\Sigma$에 대해 $U_i(U_j(t)) \neq U_j(U_i(t))$이면 U_i와 U_j는 충돌이다.

$U_i(U_j(t)) \neq U_j(U_i(t))$의 의미는 U_j 에 의해 갱신된 영역을 포함하는 갱신 연산 U_i을 적용한 후의 트리와 반대 순서로 U_i에 의해 갱신된 영역을 포함하는 갱신 연산 U_j을 적용한 후의 트리가 같지 않음을 말한다.

그림 4에서 보듯이, 갱신 연산 U_i와 U_j의 적용 영역 하부 트리의 루트를 각각 x와 y라 했을 때, $U_i(U_j(t))$ 갱신 순서에 의해 U_j에 의해 갱신된 영역은 U_i에 의해 변경되게 된다. $U_j(U_i(t))$ 갱신 순서에 의하면, U_i의 갱신 영역 일부분은 U_j에 의해 변경되게 된다. 따라서, 각각의 연산 결과에 의한 트리의 결과는 다르다 ($U_i(U_j(t)) \neq U_j(U_i(t))$).

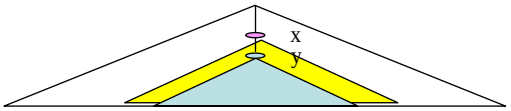


그림 4 UPDATE-UPDATE 충돌 예

3.3 효율적인 충돌 탐지 알고리즘

지금까지 READ-UPDATE와 UPDATE-UPDATE 충돌을 정의하고 개념을 소개하였다. 이번 절에서는 효율적으로 충돌을 탐지할 수 있는 알고리즘을 제안한다. 충돌 탐지 알고리즘의 기본 개념은 DTD 트리의 인코딩된 PRE(order)와 POST(order)를 이차평면으로 놓고, 연산의 (PRE, POST) 값을 이차평면에 놓았을 때 다섯 가지의 관계로 분할하여 그들의 의미를 부여하는 기법이다. 그림 5는 그림2(a)의 DTD 트리에 대해 x축은 PRE(order) 값, y축은 POST(order) 값을 단위로 표현한 이차평면을 보여주고 있으며, 이를 PRE/POST 이차평면이라 명명한다.

PRE/POST 이차평면을 통한 충돌 탐지 알고리즘에 대한 자세한 설명에 앞서, *목적노드*를 정의한다. 2.2절에

READ 연산과 UPDATE 연산은 XPath 경로 표현식을 기반으로 한다고 설명하였다. 이와 같은 XPath 경로 표현식을 통한 하부 트리의 루트를 *목적노드*라 명명한다.

정의 3 [**목적노드**] 주어진 XPath 경로 표현식에서 프레디키트를 제외한 마지막 노드를 *목적노드*라 한다.

다음은 여섯 개의 연산과 각 연산의 목적노드 및 그 목적노드의 (PRE, POST) 값에 대한 예이다. 목적노드에 대한 (PRE, POST) 값은 그림 2(b)의 PPS를 통해 얻을 수 있다. 그림 5는 각 (PRE, POST) 값을 PRE/POST 이차평면에 놓은 것을 보여주고 있다.

O1: READ(/rss/channel/item/description), 목적노드는 *description* 노드이고, 그 노드의 (PRE, POST) 값은 (5, 2) 이다

O2: REPLACE(/rss/channel/lastmodified, <latsmodified> 20070301 11:30 </lastmodified>), 목적노드는 *lastmodified* 노드이고, 그 노드의 (PRE, POST) 값은 (11, 9) 이다

O3: INSERT(/rss/channel/item, xxxx), 목적노드는 *item* 노드이고, 그 노드의 (PRE, POST) 값은 (2, 5) 이다

O4: READ(/rss//description), 목적노드는 *description* 노드이고, 그 노드의 (PRE, POST) 값은 (5, 2)와 (10, 8) 이다

O5: DELETE(/rss/channel/item[1]), 목적노드는 *item* 노드이고, 그 노드의 (PRE, POST) 값은 (2, 5) 이다

O6: REPLACE(/rss/channel/author, <author> chang </author>), 목적노드는 *author* 노드이고, 그 노드의 (PRE, POST) 값은 (12, 10) 이다

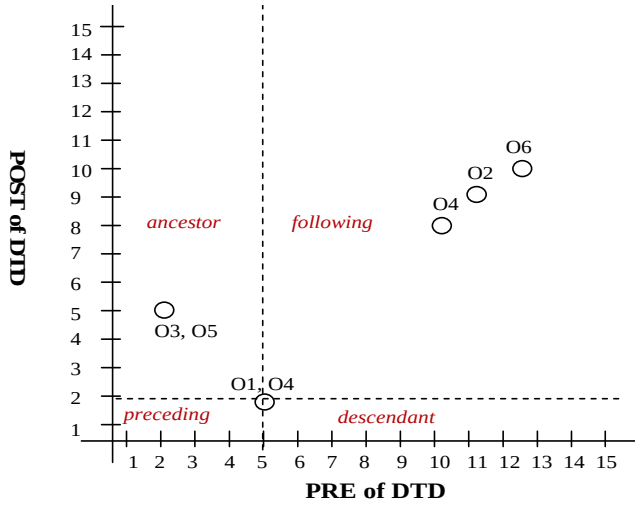


그림 5 PRE/POST 이차 평면

질의에 따라 (PRE, POST) 값은 둘 이상이 될 수 있다. 연산 O4의 목적노드에 대한 (PRE, POST) 값은 (5, 2)와 (10, 8)이다. 그러나, 연산 O1은 같은 목적노드인 *description*에 대한 (PRE, POST) 값으로 (5, 2)뿐이다. 그 이유는 O1에서 *item* 노드의 자식노드로서 *description* 노드가 목적노드이기 때문이다. 이와 같이, 목적노드의 (PRE, POST) 쌍이 집합인 경우, 불필요한 (PRE, POST) 쌍에 대한 제거는 XPath 경로 표현식의 각 노드의 PRE (POST) 값은 그 표현식의 목적노드의 PRE (POST) 값보다 작다 (크다)의 개념을 적용하는 것이다. 그림 6은 이에 대한 알고리즘을 보여준다.

Input: an XPath expression of an operation
Output: suitable (PRE, POST) values of target node of the operation
BEGIN
1. for each (Pr_{tn}, Po_{tn}) value of the target node of the operation
2. { for (Pr_{step}, Po_{step}) value of each node of the operation
3. if (!($Pr_{step} < Pr_{tn}$ and $Po_{step} > Po_{tn}$))
4. break;
5. suitable (PRE, POST) set ← (Pr_{tn}, Po_{tn})
6. }
END

Figure 6 목적노드에 대한 불필요한 (PRE, POST) 값 제거 알고리즘

지금부터 PRE/POST 이차평면의 다섯 가지 분할에 대한 의미를 설명한다. 연산 O_i와 O_j에 대한 (PRE, POST) 값을 각각 (Pr_{Oi}, Po_{Oi})와 (Pr_{Oj}, Po_{Oj})라 한다. 연산에 대한 (PRE, POST) 값을 이차평면에 놓으면 4사분면 및 자기 자신 이렇게 다섯 가지로 분할된다.

정의 4 [1사분면: O_j는 O_i에 대해 FOLLOWING 관계이다] 다음 규칙을 만족하면 O_j는 O_i에 대해 FOLLOWING 관계라 정의한다: $Pr_{Oi} < Pr_{Oj}$, $Po_{Oi} < Po_{Oj}$

정의 5 [2사분면: O_j는 O_i에 대해 PRECEDING 관계이다] 만약 다음 규칙을 만족하면 O_j는 O_i에 대해 PRECEDING 관계라 정의한다: $Pr_{Oi} > Pr_{Oj}$, $Po_{Oi} > Po_{Oj}$

정의 6 [3사분면: O_j는 O_i에 대해 ANCESTOR 관계이다] 다음 규칙을 만족하면 O_j는 O_i에 대해 ANCESTOR 관계라 정의한다: $Pr_{Oi} > Pr_{Oj}$, $Po_{Oi} < Po_{Oj}$

정의 7 [4사분면: O_j는 O_i에 대해 DESCENDANT 관계이다] 다음 규칙을 만족하면 O_j는 O_i에 대해 DESCENDANT 관계라 정의한다: $Pr_{Oi} < Pr_{Oj}$, $Po_{Oi} > Po_{Oj}$

정의 8 [O_j는 O_i에 대해 SELF 관계이다] 다음 규칙을 만족하면 O_j는 O_i에 대해 SELF 관계라 정의한다: $Pr_{Oi} = Pr_{Oj}$, $Po_{Oi} = Po_{Oj}$

그림 5에서처럼, 연산 O3과 O5은 연산 O1에 대해 ANCESTOR 관계이고, 연산 O2와 O6은 연산 O1에 대해 FOLLOWING이다. 연산 O1과 O4는 모두 READ 연산이기 때문에 고려 대상이 아니다.

READ-UPDATE 충돌 탐지. READ 연산 Ri와 UPDATE 연산 Uj의 각 목적노드의 (PRE, POST) 값을 각각 (Pr_{Ri}, Po_{Ri})와 (Pr_{Uj}, Po_{Uj})라 하면 다음 규칙 중 하나라도 만족되면 두 연산은 충돌이다:

$Pr_{Ri} > Pr_{Uj}$, $Po_{Ri} < Po_{Uj}$ (1)
(Ri는 Uj에 대해 DESCENDANT 관계이다)
$Pr_{Ri} < Pr_{Uj}$, $Po_{Ri} > Po_{Uj}$ (2)

(Ri는 Uj에 대해 ANCESTOR 관계이다)

$Pr_{Ri} = Pr_{Uj}$, $Po_{Ri} = Po_{Uj}$ (SELF) (3)

(Ri는 Uj에 대해 SELF 관계이다)

수식 (1)과 (3)인 경우 Ri는 Uj에 의해 변경된 부분의 일부분 혹은 전부를 판독하게 된다. 수식 (2)인 경우 Ri는 Uj에 의해 변경된 모든 부분을 판독하게 된다. 따라서, Ri와 Uj는 READ-UPDATE 충돌이다.

UPDATE-UPDATE 충돌 탐지. UPDATE 연산 Ui와 UPDATE 연산 Uj의 각 목적노드의 (PRE, POST) 값을 각각 (Pr_{Ui}, Po_{Ui})와 (Pr_{Uj}, Po_{Uj})라 하면 다음 규칙 중 하나라도 만족되면 두 연산은 충돌이다:

$Pr_{Ui} > Pr_{Uj}$, $Po_{Ui} < Po_{Uj}$ (4)

(Ui는 Uj에 대해 DESCENDANT 관계이다)

$Pr_{Ui} < Pr_{Uj}$, $Po_{Ui} > Po_{Uj}$ (5)

(Ri는 Uj에 대해 ANCESTOR 관계이다)

$Pr_{Ui} = Pr_{Uj}$, $Po_{Ui} = Po_{Uj}$ (6)

(Ri는 Uj에 대해 SELF 관계이다)

각 수식의 의미는 READ-UPDATE 충돌의 의미와 유사하다.

그러나, UPDATE-UPDATE 충돌 탐지 알고리즘은 완전히 직관적이지는 않다. 만약, UPDATE-UPDATE 충돌 중에 하나의 DELETE 연산과 하나의 UPDATE 연산의 충돌 관점에서만 봤을 때 DELETE-UPDATE 충돌은 보다 유연하게 완화할 수 있다. 이를 온건한 (*conservative*) DELETE-UPDATE 충돌이라 명명한다.

정의 9 [온건한 (conservative) DELETE-UPDATE 충돌] 만약 DELETE 연산 Di의 목적노드가 UPDATE 연산 Uj에 대해 ANCESTOR 관계이면, Uj를 충돌에서 무시한다.

예를 들어, 다음과 같은 하나의 DELETE 연산 D1과 하나의 REPLACE 연산 RP1이 있다고 하자.

D1: DELETE(/rss/channel/item), 목적노드 *item* 노드의 (PRE, POST) 값은 (2, 5) 이다

RP1: REPLACE(/rss/channel/item/author, <author> chang </author>), 목적노드 *author* n노드의 (PRE, POST) 값은 (6, 3) 이다

UPDATE-UPDATE 충돌 탐지 알고리즘에 의해 D1은 RP1에 대해 ANCESTOR 관계이기 때문에 충돌이다. 그러나 RP1이 D1 후에 처리된다면, D1에 의해 *item* 노드의 *author* 노드를 포함한 하부 트리가 삭제되기 때문에 RP1을 수행할 수 없게 되어 수행이 거절 된다. 만약 RP1이 먼저 수행되고 D1이 수행된다면, 두 연산은 거절 없이 수행될 것이고, 임의의 트리 t ($\in T_{\Sigma}$)에 대해 D1(RP2(t))=D1(t)이 된다. 이러한 관찰을 통해, 본 논문에서는 UPDATE 연산 (DELETE, INSERT, REPLACE)과 비교하여 ANCESTOR 관계에 있는 DELETE 연산에 대한 DELETE-UPDATE 충돌을 무시하는 온건한 (*conservative*) DELETE-UPDATE 충돌을 제안한다.

3.4 프레디키트 처리

프레디키트를 고려한 READ-UPDATE 연산 관계는 (i)프레디키트가 있는 READ-프레디키트가 없는 UPDATE, (ii)프레디키트가 없는 READ-프레디키트가 있는 UPDATE, 그리고 (iii)프레디키트가 있는 READ-프레디키트가 있는 UPDATE 이렇게 세 가지 경우가 존재한다. 첫 번째와 두 번째 경우에 있어서 프레디키트를 제외한 상태에서 두 연산이 충돌관계를 갖고 있다면, 프레디키트를 고려했을 때도 역시 충돌관계를 갖고 있다. 왜냐하면, 프레디키트를 포함한 XPath 경로 표현식에 대한 영역은 그 프레디키트를 제외한 동일한 XPath 경로 표현식에 대한 영역에 포함되기 때문이다.

그러나 세 번째 경우는 다르다. 예를 들어, 다음과 같이 하나의 READ 연산 R1과 하나의 REPLACE 연산 RP2가 있다고 하자.

R1: READ(/rss/channel/item[title="xxxx"]), 목적노드 *item* 노드의 (PRE, POST) 값은 (2, 5) 이다

RP2: REPLACE(/rss/channel/item[title="yyyy"]/author, <author> chang </author>), 목적노드 *author* 노드의 (PRE, POST) 값은 (6, 3) 이다

READ-UPDATE 충돌 탐지 알고리즘에 의해 R1은 RP2에 대해 ANCESTOR 관계이기 때문에 충돌이다. 그러나 R1의 프레디키트에 의한 하부 트리와 RP1의 프레디키트에 의한 하부 트리에 대한 공통 부분은 없다. 왜냐하면 두 연산 모두 동일하게 *item* 노드의 자식 노드인 *title*에 대해 프레디키트를 갖고 있는데, 그 값이 다르기 때문이다. 즉, RP2의 title 값이 "yyyy"인 곳의 author 엘리먼트 내용을 교체한 결과를 R1에 의해 판독되지 않는다. 따라서, 두 연산은 충돌 관계가 없다. 이러한 관찰을 통해 다음과 같이 프레디키트를 갖고 있는 두 연산의 충돌 탐지 알고리즘을 제안한다.

설명의 단순함을 위해 다음과 같은 함수를 정의한다:

- PREDICATES(*o*): 임의의 연산 *o*에서의 모든 값 기반 프레디키트 집합
- VALUE(*p*): 임의의 값 기반 프레디키트 *p*에서의 에서의 값 출력
- (Pr_p, Po_p): 임의의 프레디키트 *p*에서 그 프레디키트의 목적노드의 (PRE, POST) 값

프레디키트를 갖고 있는 READ-UPDATE 충돌 탐지. 프레디키트를 갖고 있는 READ 연산 R_i과 역시 프레디키트를 갖고 있는 UPDATE 연산 U_j에 대해 프레디키트를 제외했을 때 두 연산은 충돌 관계를 가지고 있다고 하자. 만약 다음의 조건을 만족하면 두 연산은 충돌이 아니다. 그렇지 않으면 두 연산은 충돌 관계이다:

- R_i와 U_j의 모든 프레디키트 유형은 값-기반 프레디키트이다.
- Ri의 각각의 프레디키트는 U_j의 각각의 프레디키트와

SELF 관계이면서 value 값이 다르다.

$\forall$ pi $\in$ PREDICATES(R_i), $\forall$ qj $\in$ PREDICATES(U_j) $\Rightarrow$
($Pr_{pi} = Pr_{qj}$, $Po_{pj} = Po_{qj}$) $\wedge$ (VALUE(pi) $\neq$ VALUE(qj))

이 알고리즘은 XML 인스턴스 정보가 없는 선처리 방법이기 때문에 모든 가능한 충돌을 탐지한다. 향후 보다 나은 최적의 충돌 탐지 방법을 연구하고자 한다.

프레디키트가 있는 UPDATE-UPDATE 연산 사이의 충돌 역시 유사한 방법으로 탐지된다.

4장 결론

본 논문에서는 유효한 XML 환경에서 PRE/POST 스키마 구조 정보 및 PRE/POST 이차평면을 이용한 READ-UPDATE 및 UPDATE-UPDATE 연산 사이의 효율적인 충돌 탐지 기법을 제안하였다. 제안하는 기법은 XML 인스턴스 정보가 필요 없고, 질의 처리기에 독립적인 선처리 방법으로써 어떠한 유효한 XML 데이타베이스 관리 시스템에 적용할 수 있는 장점을 갖고 있다.

갱신 연산을 지원하는 데이터베이스 관리 시스템에서 연산 사이의 충돌 탐지는 질의 컴파일러의 질의 최적화 기법 및 트랜잭션 관리기의 병행수행 스케쥴링 기법에 필수적이기 때문에 논문에서 제안하는 기법은 유효한 XML 데이타베이스 시스템에서 이들에 대한 개발에 유용하게 사용될 수 있다.

지금까지 설명한 충돌 탐지 기법은 연산 대 연산의 정적 충돌만을 고려하고 있다. XML에서의 갱신 연산은 구조 정보뿐만 아니라 인스턴스 정보 (예를 들어, “?”, “*”, “+”) 역시 바꾸기 때문에 다른 연산에 영향을 준다. 즉, 다른 갱신 연산에 의해 유효한 갱신 연산이 유효하지 않을 수 있고, 유효하지 않던 갱신 연산이 유효할 수 있다. 이와 같은 동적 환경의 충돌 문제가 발생한다. 추후 연구로는 동적 환경의 충돌 문제를 유효한 XML 환경의 병행수행 제어 기법을 통해 연구하고자 한다.

References

[1] I. Tatarinov, Z. G. Ives, A. Y. Halevy, and D. S. Weld, "Updating XML", In Proc. ACM SIGMOD Int. Conf. on Management of Data, pp. 413-424, 2001.

[2] M. Rys, “Proposal for an XML Data Modification Language”, Microsoft Reopt, 2002.

[3] D. Barbosa, A. O. Mendelzon, L. Libkin, L. Mignet, and M. Arenas, “Efficient Incremental Validation of XML Documents”, In Proc. the 20th Int. Conf. Data Engineering (ICDE), pp. 671-682, 2004.

[4] B. Kane, H. Su, and E. A. Rundensteiner, “Consistently updating XML documents using incremental constraint check queries”, Proc. the 4th Int. workshop on Web Information and Data Management (WIDM), pp. 1-8, 2002.

[5] B. Bouchou and M. H. F. Alves, “Updates and Incremental Validation of XML Documents”, 9th Int. Conf. on Database Programming Languages (DBPL), pp. 216-232, 2003.

[6] A. Balmin, Y. Papakonstantinou, and V. Vianu. Incremental validation of XML documents. ACM Transsaction on Database Systems (TODS), 29(4), pp. 710-751, 2004

[7] D. Barbosa, G. Leighton and A. Smith D., “Efficient Incremental Validation of XML Documents After Composite Updates”, In Proc. the Fourth Int. XML Database Symposium (XSym), pp. 107-121, 2006.

[8] Extensible Markup Language (XML) 1.0, second edition, http://www.w3.org/TR/2000/REC-xml-20001006

[9] XQuery 1.0 and XPath 2.0 Full-Text, http://www.w3.org/TR/xquery-full-text/

[10] S. Helmer, C. C. Kanne, and G. Moerkotte, "Evaluating Lock-based Protocols for Cooperation on XML Documents", ACM SIGMOD Record, Vol 33, No.1, pp. 58-63, 2004.

[11] T. Grabs, K. Böhm, and H. J. Schek, "XMLTM: Efficient Transaction Management for XML Documents", In Proc. the 13th ACM Conf. Information and Knowledge Management (CIKM), pp. 142-152, 2002.

[12] S. Dekeyser, J. Hidders, and J. Paredaens, "A Transaction Model for XML Databases" World Wide Web Journal, Kluwer, pp. 1- 36, 2003.

[13] M. Raghavachari and O. Shmueli, “Conflicting XML Updates”, the 10th Int. Conf. Extending Database Technology (EDBT), pp. 552-569, 2006.

[14] T. Grust, “Accelerating XPath Location Steps”, In Proc. of the 21st Int’l ACM SIGMOD Conf. on Management of Data, pages 109-120, 2002.

[15] S. Mohan, A. Sengupta, Y. Wu, and J. Klinginsmith, “Access Control for XML- A Dynamic Query Rewriting Approach”, In Proc. of the 14th ACM Conference on Information and Knowledge Management (CIKM), pp. 251-252, 2005.