XML 데이타베이스 변경 연산의 즉시 부분 검증 메카니즘

(Immediate and Partial Validation Mechanism for Update Operations in XML Databases)

김 상 균 † 이 규 철 ††
(Sang-Kyun Kim) (Kyu-Chul Lee)

요 약 최근에 데이타베이스에 저장된 XML문서의 변경에 대한 여러 연구들이 수행되었다[1-3]. 이 연구들은 우선 변경 연산들을 정의하고 이 연산들을 수행할 때 발생하는 의미적 문제들을 해결하는 방법을 제안하였다. 이러한 연구들에서는 변경 연산을 수행한 후에 검증을 수행하기 때문에 여러 가지 충돌 문제가 발생한다. 이러한 충돌문제를 해결하기 위해서는 XML데이타베이스 시스템이 변경 연산을 수행하기 전에 이 연산이 DTD에 따르는지 즉시 검증할 수 있어야 한다.

또한 지금까지의 연구들은 변경된 XML문서를 검증할 때 변경된 부분을 검증하지 않고 문서 전체를 검증한다. 따라서 많은 응용프로그램들이 XML문서를 변경한다면 이 연산에 대한 검증 과정 때문에 심각한 성능 저하를 가져올 것이다.

이 연구에서는 이 두 가지 문제점을 해결하기 위한 방법을 제안한다. 우선 DTD정보를 추출하여 데이타베이스에 저장하고 데이타베이스에 저장된 XML문서를 변경할 때 이 저장된 DTD정보를 이용하여 변경 연산의 유효성을 검증함으로써 항상 유효한 XML문서를 데이타베이스에 유지하도록 한다. 이를 위해 유효성 검증을 변경 연산이 수행되기 전에 즉시 수행하고 검증 범위 또한 변경된 부분으로 한정시키는 메카니즘을 고안하였다.

키워드 : XML, DTD, 검증, 변경 연산

Abstract Recently, several works have been proposed for updating XML documents[1-3] stored in databases. These researches defined update operations and resolved some semantic problems. Because the update operations are usually validated after execution, several conflicts may occur. For solving these conflicts, XML database systems must be able to validate an update operation immediately according to DTD before the update operation is executed.

Furthermore, in many studies for updating, they just validate whole XML documents and can't validate parts of them. If updates are very frequent, validating whole XML documents will cause performance degradation.

In this paper, we propose solutions for these two problems. We extract and store DTD information. Then, when an XML document stored in the database is updated, we verifies whether the update is valid or not by using the information. Consequently, XML database systems can always maintain valid XML documents. The validity of update operations is checked immediately before the actual update operation is applied to the database and the validation is performed on only updated parts of an XML document in the database

Key words : XML, DTD, Validation, Update Operation

† 비 회 원 : 충남대학교 컴퓨터공학과
skkim@ce.cnu.ac.kr
†† 종신회원 : 충남대학교 컴퓨터공학과 교수
kclee@ce.cnu.ac.kr
논문접수 : 2002년 5월 16일
심사완료 : 2003년 4월 9일

1. 서 론

XML이 인터넷 전자 문서 표준으로 제정됨에 따라 많은 데이타베이스 회사나 학교, 연구소등에서 표준화된 데이타 교환 포맷으로 XML을 이용하려는 연구가 이루

어지고 있다. XML전용 데이타베이스 시스템들은[4,5] XML을 효율적으로 관리하기 위한 하부 구조를 가지고 있으며, 많은 상용 데이타베이스 시스템들[6-8] 또한 기존의 데이타베이스 기능에 XML을 저장, 검색하기 위한 기능들을 추가적으로 제공하고 있다. 또한 최근에는 기존 관계 데이타베이스에 저장된 데이타를 XML 데이타와 같이 처리해야할 필요가 증가함에 따라 관계 데이타베이스에 저장된 데이타를 XML로 내보내기 위해 XML 뷰를 이용하는 방법들이 연구되고 있다[9,10].

하지만 XML데이타베이스 시스템에서 XML을 처리하기 위해서 XML문서를 저장, 검색하는 기능뿐만 아니라 XML문서를 변경하는 기능도 필요함에 따라 최근 XML문서의 변경에 대한 몇몇 연구들이 제안되었다. 이 연구들에서는 변경 연산에 대한 구문들을 정의하고, 이 연산들을 수행할 때 발생하는 의미적 문제들에 대한 해결 방법을 제안하고 있다. 하지만 변경연산을 수행할 때 이 연산이 DTD에 따라 유효한지 검증하지 않고 변경 연산들이 수행된 후에 검증되기 때문에 여러 가지 문제가 발생한다. 특히 하나의 변경문에서 여러 개의 변경 연산을 수행할 때 이 연산들 사이에 충돌이 발생하게 된다.

이를 막기 위해서는 데이타베이스에 저장된 XML문서를 변경할 때 XML데이타베이스 시스템이 변경 연산을 수행하기 전에 이 연산이 DTD에 따르는지 검사해서 유효하면 변경 연산을 수행하고 아니면 수행하지 않아야 한다.

일반적으로 파서는 문서 전체를 단위로 검증(i.e. Full Validation)하며 문서의 일부분을 파싱하고 브라우징(i.e. Partial Validation)[11]할 수 없다. 또한 어플리케이션에서 문서의 일부분을 변경하였을 때 변경된 XML문서가 유효한지 검사하려면 문서 전체를 다시 파싱해야 한다. 이러한 과정은 문서의 크기가 커지면 커질수록 상대적으로 엘리먼트 개수가 많아지게 되고 문서 전체를 파싱하는 시간이 오래 걸리게 되어 매우 비효율적이다.

더구나 XML파일이 아니라 데이타베이스에 저장되어 있는 XML문서에 대해서는 문제가 더욱 심각하다. 데이타베이스에 저장되어 있는 XML문서가 변경되었을 때, 이를 검증하기 위해서는 DTD와 XML 문서 전체를 꺼내서 파싱하고 유효성을 검사한 후 유효하지 않으면 다시 저장해야만 한다. 만약 이런 상황에서 데이타베이스에 저장된 XML문서에 대한 변경 연산이 빈번하게 일어난다면 문서 전체를 꺼내고 파싱해서 검증한 후에 이를 다시 저장하는 시간 때문에 심각한 성능 문제를 겪게 된다.

이러한 성능 문제를 해결하기 위해서는 데이타베이스에 저장된 XML문서에 대해 변경 연산이 일어났을 때 XML문서 전체를 다시 파싱하는게 아니라 변경된 부분만 검증할 수 있는 방법이 필요하다.

본 연구에서는 위의 두 가지 문제를 해결하기 위해서 데이타베이스에 저장된 XML문서에 대한 변경 연산이 일어났을 때 XML저장 관리 시스템에서 변경된 부분만 즉시 검증해 줄 수 있는 방법을 제안하고자 한다. 이 메카니즘은 우선 DTD정보를 추출하여 데이타베이스에 저장하고 데이타베이스에 저장된 XML문서를 변경할 때 이 저장된 DTD정보를 이용하여 변경 연산의 유효성을 검증함으로써 항상 유효한 XML문서를 데이타베이스에 유지하도록 한다. 따라서 이 연구는 DTD가 존재하는 문서의 변경에 대해 검증을 고려하였다.

본 논문의 구성은 다음과 같다. 2장에서는 XML 데이타베이스에서 변경 연산의 충돌 문제를 살펴보고, 3장에서는 이를 해결하기 위해 변경 연산시에 즉시 부분 검증을 지원할 수 있는 메카니즘을 설명한다. 4장에서는 이 메카니즘에 대한 성능을 평가하고 마지막으로 5장에서 결론을 맺는다.

2. 관련 연구

XML문서의 변경에 대한 연구들은 XQuery[12]를 사용하여 변경 연산의 구문을 정의하고, 이 연산들을 수행할 때 발생하는 의미적 문제들을 해결하는 방법들을 제안하고 있다. 하지만 하나의 변경문에서 여러 개의 변경 연산들이 실행되면 이들 사이에 충돌이 발생할 수 있다.

그림 1은 Robie[2]가 XQuery를 이용하여 XML문서를 변경하는 예제 중의 하나로서 이 변경문은 bib 엘리먼트 밑에 book 엘리먼트를 삽입한 후에 author 엘리먼트를 삽입한다. 하지만 왼쪽의 DTD에 따르면 author가 먼저 삽입되어야 한다. Robie[2]는 여러 가지 충돌 문제를 해결하기 위해서 하나의 노드에 대해 두 번 이상 삭제와 변경연산을 할 수 없도록 하는 방법을 제안했지만, 그림 1과 같이 엘리먼트 삽입 순서에 따른 충돌 문제가 발생하며, 이러한 문제를 해결하기 위해서 변경 연산을 수행한 후에 파서를 통해 전체 결과를 검증하고 있다.

Tatarinov[1]는 모든 변경 연산들이 유효하다고 가정하기 때문에 사용자가 유효하지 않은 연산을 수행하기 전까지는 이러한 문제가 발생하지는 않는다. 하지만 이러한 경우는 충분히 발생할 수 있기 때문에 이 문제를

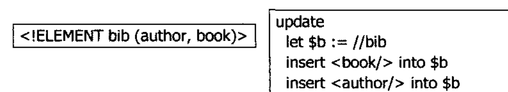

그림 1 변경연산 수행시 충돌이 발생하는 예제

해결할 방법이 필요하다. 또한 변경 연산을 지원하는 XML데이타베이스 시스템으로 eXcelon[4]이 있는데, 이 시스템은 윈도우 창에서 편집기를 통해 변경 연산을 수행하며 변경 연산에 대한 검증을 수행하지 않는다. 대신 문서 전체를 검증하기 위한 메뉴를 두어서 사용자가 변경 연산을 수행한 후에 이 메뉴를 선택하면 이미 변경된 XML문서 전체에 대해 검증을 수행한다.

위와 같은 연구들은 모두 변경 연산을 수행한 후에 파서를 이용해서 변경된 XML문서 전체를 검증하는 방법을 사용하고 있다. 이러한 방법을 사용하는 이유는 현재까지 XML데이타베이스에 저장된 XML문서의 부분만 검증할 수 있는 메카니즘이 존재하지 않기 때문이다. 따라서 유효성을 검증하기 위해서는 XML문서 전체를 검증해야 하는데 커다란 문서일수록 그리고 변경 연산의 자주 발생할수록 전체 문서의 검증시간에 대한 오버헤드 때문에 변경 연산을 수행하기 전에 이 연산의 유효성을 검증하지 않는다.

빠른 변경 연산을 요구하는 XML 데이타베이스 시스템들은 이러한 이유 때문에 변경 연산 수행 전에 검증을 하지 않으며 만약 사용자가 항상 유효한 변경 연산을 수행하지 않으면 다른 사용자가 변경된 문서를 검색할 때 유효하지 않은 검색결과를 낳게 된다.

이러한 문제를 해결하기 위해 이 연구에서는 변경 연산을 수행하기 전에 즉시 변경된 부분만 검증할 수 있는 메카니즘을 설명한다. 4장에서는 이 메카니즘을 사용한 시스템이 다른 시스템보다 변경 연산을 검증할 때 항상 좋은 성능을 가지는 것을 보일 것이다.

3. XML 즉시 부분 검증 메카니즘

XML문서는 구조적인 정보를 가진다. 따라서 XML문서를 데이타베이스에 저장하거나 응용 프로그램에서 사용하기 전에 파서에 의해서 이 구조적인 정보가 유효한지 검증되어야 한다. 또한 XML문서가 변경되었을 때 이 문서의 전체 또는 부분을 다시 검증해야 한다. 다음은 이러한 XML문서의 검증을 검증 단위와 검증 방법을 기준으로 나눈 것이다.

- 검증 단위
  - 전체 검증(Full validation) : 전체 XML문서가 검증되어야 한다.
  - 부분 검증(Partial validation) : XML문서의 변경된 부분만 검증된다.
- 검증 방법
  - 지연 검증(Deferred validation) : 변경 연산 실행 후 검증 연산 수행
  - 즉시 검증(Immediate validation) : 변경 연산이 수행되기 전에 검증 후 유효하면 변경

현재 대부분의 XML데이타베이스 시스템은 저장된 XML문서가 변경되었을 때 *지연 전체 검증 방법*을 사용하고 있다. 이 방법은 서론에서 언급했듯이 변경 연산을 수행할 때 충돌이 발생할 수 있으며 변경 연산에 대한 성능도 저하되는 문제점이 있다.

따라서 이러한 문제를 해결하기 위해서는 변경 연산이 일어났을 때 XML 저장관리 시스템이 변경된 부분을 즉시 검증할 수 있는 *즉시 부분 검증 방법*을 사용해야 한다. 이를 통해 데이타베이스에 저장된 XML문서의 유효성을 유지할 수 있고, 변경된 부분만 검증하므로 전체 검증보다 성능이 좋다.

XML문서를 검증하기 위해서는 DTD파일이 필요하다. 마찬가지로 XML데이타베이스에서 즉시 부분 검증을 지원하기 위해서는 DTD정보가 저장되어 있어야 한다. 하지만 부분 검증을 위해서 DTD는 파일 형태가 아닌 어떠한 파싱된 정보로 저장되어야 하며, 변경 연산이 일어났을 때, 이 정보를 보고 빠르게 검증할 수 있어야 한다. 이러한 XML데이타베이스에서의 변경 연산을 검증하기 위한 과정은 다음과 같다.

(1) DTD를 저장시 DTD를 파싱한 후 DTD 정보를 추출
(2) 추출한 DTD정보를 데이타베이스에 저장
(3) 변경 연산 수행시 저장된 DTD정보를 참조하여 유효성 판단
(4) 유효하면 변경 연산을 실행하고 유효하지 않으면 변경 연산 취소

다음 절에서는 이러한 부분 검증을 위한 DTD 정보 추출 및 저장 방법에 대해 연구하고 이 정보를 이용해서 검증하는 과정에 대해 설명한다.

3.1 DTD 정보 추출

이 절에서는 즉시 부분 검증을 위해 필요한 DTD정보를 추출하는 방법을 설명한다. 즉시 부분 검증을 위해 필요한 DTD정보는 보통의 파서에서 만들어지는 것과 다르다. 따라서 일반 파서 안에 즉시 부분 검증을 위한 모듈을 추가하였다. 이 모듈은 DTD 파일에서 엘리먼트 선언과 애트리뷰트 선언 정보를 추출한다.

DTD에서 똑같은 이름을 가지는 엘리먼트 선언은 발생할 수 없다. 따라서 한 엘리먼트가 변경되었을 때, 이 엘리먼트를 검증하려면 우선 이 노드의 부모 노드에 대한 엘리먼트 선언을 찾고 변경된 노드가 유효한지 검사하면 된다. 이 때 한 엘리먼트의 부모는 하나밖에 없고 부모 노드는 쉽게 찾을 수 있다.

이러한 엘리먼트 선언들은 각각 하나의 정규식으로 인식될 수 있다. 따라서 정규식을 인식할 수 있는 결정적 유한 오토마타(Deterministic Finite Automata, DFA)[14]를 사용하면 DTD정보를 쉽게 구성할 수 있

다. 우선 하나의 엘리먼트 선언마다 하나의 DFA를 구성한다. 이렇게 하면 위에서 말한 바와 같이 하나의 엘리먼트가 변경되었을 때 부모 노드의 엘리먼트 선언에 대한 DFA를 찾고, 그 DFA에서 변경된 엘리먼트의 유효성을 검사하면 된다.

이 때 변경된 엘리먼트의 유효성을 검사하기 위해서는 부모 엘리먼트의 모든 자식들을 검사할 필요는 없으며 변경된 엘리먼트와 형제 엘리먼트와의 관계가 유효한지만 검사하면 된다. 즉, 변경된 엘리먼트와 변경된 엘리먼트의 앞과 뒤의 형제 엘리먼트 이렇게 기껏해야 3개의 엘리먼트가 검증과정에서 필요하다.

DTD에서 엘리먼트를 선언할 때, 엘리먼트 이름 외에 ','(sequence), '|'(choice), '?'(1 or 0), '*'(0 or more), '+'(1 or more) 이렇게 5가지 연산자와 괄호를 이용한다. DTD를 파싱하여 DFA를 구성하기 위해서는 본 연구에서는 그림 2와 같이 각 연산자마다 하나의 패턴을 만들고 연산자가 나올 때마다 그것들을 조합하여 구성한다. 이 경우 만약 엘리먼트 내용 모델이 "EMPTY" 또는 "ANY"로 선언되어 있는 경우에는 입력 즉, 전이 엘리먼트 이름을 "EMPTY"와 "ANY"로 저장한다.

괄호는 괄호 안의 DFA가 또 하나의 패턴을 이룰 수

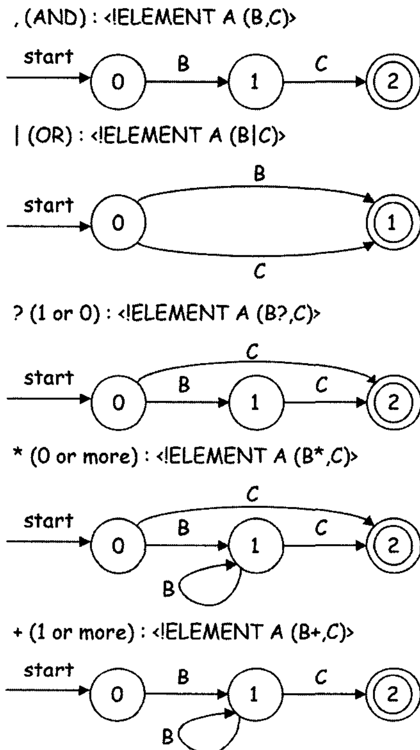

그림 2 각 연산자에 대한 패턴 구성

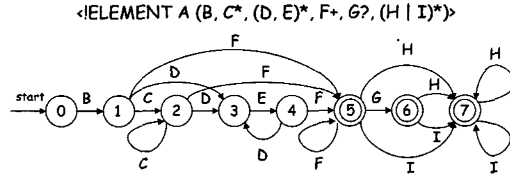

그림 3 하나의 엘리먼트 선언에 대한 DFA 표현

있기 때문에 재귀적으로 처리한다. 그리고 만약 삭제할 엘리먼트가 '+'(1 or more)로 선언되었다면 XML문서에서 적어도 하나 이상의 엘리먼트가 존재해야 한다. 따라서 한 부모의 인접한 자식들 중에서 똑같은 이름을 가지는 엘리먼트 개수가 한 개일 경우 이를 삭제하려고 하는 연산은 유효하지 않게 된다. 이 연구에서 제안한 방법에서는 대부분 하나의 DFA에서 '+'로 선언된 엘리먼트의 이전 엘리먼트에서 다음 엘리먼트로의 전이가 일어날 수 없기 때문에 문제가 되지 않는다. 하지만 '+'로 선언된 엘리먼트가 마지막 엘리먼트일 경우 다음 엘리먼트가 존재하지 않기 때문에 이를 결정할 수 없게 된다. 이를 위해서 DFA의 마지막 상태를 이용한다. 즉, 삭제하려는 엘리먼트가 마지막 엘리먼트이고, 삭제하려는 엘리먼트와 똑같은 이름을 가지는 이전 엘리먼트가 없을 때, 현재 상태가 마지막 상태이면 삭제할 수 없게 된다.

애트리뷰트 선언은 엘리먼트 이름, 애트리뷰트 이름, 애트리뷰트 값, 애트리뷰트 타입, 디폴트 타입, 디폴트값 이렇게 6가지 요소로 구성된다. 따라서 엘리먼트 선언과 같이 특별히 파싱된 형태를 가질 필요는 없고 각각의 요소를 분리해서 추출하면 된다.

3.2 DTD 정보 저장

3.2.1 DTD 정보 저장 스키마

추출한 DTD정보는 변경 연산을 검증할 때 사용해야 하기 때문에 데이타베이스에 저장해야 한다. 따라서 기존의 XML저장 관리 시스템에 추가적인 저장 구조가 필요하다. 본 연구에서는 엘리먼트와 애트리뷰트 정보를 저장하기 위한 두 개의 테이블을 추가하고 이 테이블에 추출한 DTD정보를 저장한다. 이 두 테이블의 스키마는 그림 4와 같다.

ELEDEFTB	
dtd_id	varchar2(5)
elementName	varchar2(100)
beforeState	number(7)
tranName	varchar2(100)
afterState	number(7)
finalState	number(1)

ATTDEFTB	
dtd_id	varchar2(5)
elementName	varchar2(100)
attrType	varchar2(20)
attrName	varchar2(100)
attrValue	varchar2(200)
defaultType	varchar2(10)
defaultValue	varchar2(100)

그림 4 DTD 정보 저장 테이블

- ELEDEFTB : 파서에서 엘리먼트 선언을 파싱하여 DFA 형태로 만들고 이를 저장하기 위한 테이블이며 다음과 같은 애트리뷰트를 가진다.
  - dtd_id : 파싱한 DTD의 id
  - elementName : 하나의 엘리먼트 선언에서의 부모 엘리먼트 이름
  - beforeState : 엘리먼트가 출발하는 상태
  - tranName : 전이하는 엘리먼트의 이름
  - afterState : 엘리먼트가 도착하는 상태
  - finalState : 엘리먼트가 마지막 상태인가를 기록
- ATTDEFTB : 애트리뷰트 선언은 다음과 같은 애트리뷰트를 가진다.
  - dtd_id : 파싱한 DTD의 id
  - elementName : 애트리뷰트가 속한 엘리먼트의 이름
  - attrName : 애트리뷰트 이름
  - attrType : 애트리뷰트의 타입(CDATA, ID, ENTITY 등)
  - attrValue : 애트리뷰트의 값
  - defaultType : 디폴트 타입(REQUIRED, FIXED, DEFAULT)
  - defaultValue : FIXED나 DEFAULT로 선언되었을 때 디폴트 값

3.2.2 DFA형태의 엘리먼트 선언 저장

변경된 파서를 통해서 하나의 엘리먼트 선언이 파싱되면 하나의 엘리먼트 선언마다 하나의 DFA가 생성된다. 이것을 테이블에 삽입하기 위해서 우선 모든 전이를 분리한다. 그러면 각각의 전이는 시작 상태, 끝 상태, 전이 엘리먼트 이름, 마지막 상태를 가지는데 이것을 dtd_id, elementName과 함께 저장하면 된다. 여기에서 dtd_id는 DFA가 어떤 DTD에 속하는지를 나타내며, elementName은 DFA가 어떤 엘리먼트 선언에 속하는지를 나타내기 위해서 엘리먼트 선언 이름을 같이 저장한다. 그림 5는 그림 3과 같은 DFA를 데이타베이스에 저장하는 예제이다.

dtd_id	elementName	beforeState	tranName	afterState	finalState
paper	A	0	B	1	0
paper	A	1	C	2	0
paper	A	1	D	3	0
paper	A	1	F	5	1
paper	A	2	C	2	0

그림 5 그림 3의 DFA를 ELEDEFTB 테이블에 저장하는 예제

3.2.3 애트리뷰트 선언 저장

모든 애트리뷰트 선언은 6개의 요소로 분리되어 추출

<!ATTLIST eg xml:space (default|preserve) #FIXED preserve >
<!ATTLIST loc href CDATA #REQUIRED>

dtd_id	element Name	attrName	attrType	attrValue	default Type	default Value
paper	eg	xml:space	Enumeration	default| preserve	FIXED	preserve
paper	loc	href	CDATA		REQUIRED	

그림 6 애트리뷰트를 ATTDEFTB에 저장하는 예제

될 수 있다. 따라서 애트리뷰트를 저장할 때는 각각의 요소를 dtd_id와 함께 테이블에 저장하면 된다. 그림 6은 애트리뷰트를 데이타베이스에 저장하는 예제이다.

3.3 변경 연산 유효성 검증

3.3.1 변경 연산의 종류와 검증

엘리먼트와 애트리뷰트에 표 1과 같은 변경 연산을 통해 변경을 수행하기 전에 다음과 같은 검증 과정을 거친다. 만약 이 과정을 만족하면 변경 연산을 수행하게 되고, 만족하지 못하면 수행하지 않고 오류를 발생하게 된다.

이 연구에서는 XML문서를 접근하고 관리하기 위해 W3C에서 만든 표준인 DOM Level 1[13]에 나오는 변경 연산에 대해서 이 연구의 검증 방법을 적용하였다. DOM Level 1에서는 엘리먼트와 애트리뷰트만 변경이 가능하기 때문에 이 연구에서도 이 두 가지 노드를 변경할 경우에 대한 검증을 고려하며 추출한 DTD정보나 다른 노드들에 대한 변경은 없는 것으로 가정한다.

표 1은 DOM Level 1에 나오는 변경 연산들을 정리한 것이다. 이 함수들은 기본적으로 하나의 노드를 삽입하고 삭제하지만 만약 서브트리를 가지는 엘리먼트를 삽입할 경우는 서브트리를 순회하면서 삽입과정을 반복하게 되고, 서브트리를 삭제할 경우는 서브트리안의 엘리먼트에 ID가 있는지 검사한 후 모든 하위 노드를 삭제하게 된다. 만약 해당 ID에 대한 참조가 있는 경우 삭제할 수 없다.

3.3.2 엘리먼트 검증 과정

- 엘리먼트 삽입

엘리먼트가 삽입될 때의 검증 과정은 그림 7의 알고리즘을 따른다. 엘리먼트를 삽입하려면 우선 DTD에 선언되어 있는 엘리먼트인지, 부모 엘리먼트가 "ANY"로 선언되어 있는지 검사해야 한다. DTD에 선언되지 않은 엘리먼트일 경우에는 삽입될 수 없으며, 부모 엘리먼트가 "ANY"로 선언되어 있으면 어떤 엘리먼트가 자식 엘리먼트로 와도 상관이 없기 때문에 유효한 연산이 된다. 이러한 과정을 거친 후 삽입하는 엘리먼트가 삽입하는 곳의 이전 엘리먼트와 이후 엘리먼트 사이에서 형

표 1 DOM Level 1에 나오는 변경 연산

클래스	함 수	설 명
Node	appendChild(Node newChild)	newChild 노드를 마지막 자식으로 삽입
	insertBefore(Node newChild, Node refChild)	refChild 앞에 newChild를 삽입
	removeChild(Node oldChild)	oldChild를 삭제
	replaceChild(Node newChild, Node oldChild)	oldChild를 newChild로 바꿈
NamedNodeMap	removeNamedItem(DOMString name)	이름이 name인 애트리뷰트를 NamedNodeMap에서 삭제
Attr	setValue(DOMString value)	애트리뷰트 값을 value로 설정
Element	removeAttribute(DOMString name)	이름이 name인 애트리뷰트 삭제
	removeAttributeNode(Attr oldAttr)	oldAttr 애트리뷰트 삭제
	setAttribute(DOMString name, DOMString value)	이름이 name인 애트리뷰트가 존재하면 교체, 없으면 value값을 가지는 애트리뷰트 추가
	setAttributeNode(Attr newAttr)	newAttr 애트리뷰트가 존재하면 교체, 없으면 추가

Algorithm1 : 엘리먼트 삽입 검증
Input : 부모 엘리먼트 *parentX*, 삽입할 곳 바로 이전 엘리먼트 *previousX*, 삽입할 엘리먼트 *X*, 삽입할 곳 바로 다음 엘리먼트 *nextX*
Output : 유효하면 "yes" 아니면 "no" 리턴
Steps :

```
if X가 DTD에 선언 then
   if parentX의 내용 모델 == "ANY" then return "yes";
   else
      if previsouX와 똑같은 이름을 가지는 엘리먼트가 두 개 이상 then
         previousX := parentX의 첫 번째 자식;  X := previousX의 next sibling;
         while parentX의 모든 자식 do
            if insertValidationProcess(previousX, X, null) == false then return "no";
            previousX := X;  X := X의 next sibling;
         end
         return "yes";
      end if
      else return insertValidationProcess(previousX, X, nextX);
   end else
end if
else return "no";
```

Algorithm2 : insertValidationProcess (삽입시 DFA에서 전이 가능 검사)
Input : 이전 엘리먼트 *previousX*, 엘리먼트 *X*, 다음 엘리먼트 *nextX*
Output : 전이가 존재하면 "yes" 아니면 "no" 리턴
Steps :

```
if previousX 다음에 X로 전이 가능 then
   if nextX가 존재 then
      if X 다음에 nextX로 전이 가능 then return "yes";
      else return "no";
   end if
   else return "yes";
end if
else return "no";
```

그림 7 엘리먼트 삽입 검증에 대한 알고리즘

제 관계를 이룰 수 있는지 검사한다. 이것을 검증하기 위해서는 우선 삽입하는 곳의 이전 엘리먼트를 찾고 이 엘리먼트 다음에 삽입하는 엘리먼트가 올 수 있는지 그리고 삽입하는 엘리먼트 다음에 삽입되는 곳의 다음 엘리먼트가 올 수 있는지 검사해야 한다. 이 때 삽입되는 곳의 다음 엘리먼트가 없으면 마지막 검사는 생략할 수 있다.

다시 말해서 그림 3에서 하나의 엘리먼트 선언에 대해 DFA를 구성한 예를 보면 모든 엘리먼트는 하나의 상태로 도착한다. 따라서 삽입하려는 곳의 이전 엘리먼트가 끝나는 상태를 찾는다. 이 상태에서 삽입하는 엘리먼트가 전이될 수 있고, 삽입하는 엘리먼트가 끝나는 상태에서 삽입하려는 곳의 다음 엘리먼트가 전이할 수 있으면 유효한 삽입 연산이 된다.

하지만 만약 하나의 엘리먼트 선언에서 삽입하는 곳 이전 엘리먼트와 똑같은 이름의 엘리먼트가 2개 이상 선언되어 있으면 이 엘리먼트를 찾기가 힘들어진다. 따라서 이 경우에는 자식들 모두를 처음부터 검사해야 한다.

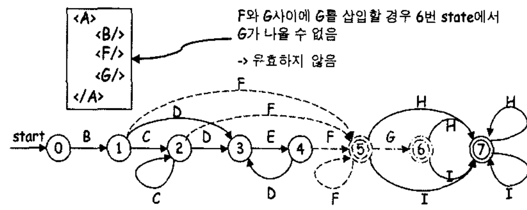

그림 8 엘리먼트 삽입 연산에 대한 검증

다음은 하나의 엘리먼트가 삽입되었을 때 위의 조건에 따라 검증하는 과정을 예로 보인 것이다.

(가) G는 DTD에 선언된 엘리먼트이다.

(나) 부모 엘리먼트인 A는 “ANY”로 선언 되어 있지 않다.

(다) 삽입할 곳 바로 전 엘리먼트의 도착 상태는 5번 하나이다.

(라) 삽입하려는 엘리먼트 G는 5번 상태에서 6번 상태로 전이가 일어날 수 있다.

(마) 현재 상태 6번에서는 삽입할 곳 바로 다음 엘리

Algorithm1 : 엘리먼트 삭제 검증
Input : 부모 엘리먼트 *parentX*, 삭제할 곳 바로 이전 엘리먼트 *previousX*, 삭제할 엘리먼트 *X*, 삭제할 곳 바로 다음 엘리먼트 *nextX*
Output : 유효하면 “yes” 아니면 “no” 리턴
Steps :

```
if parentX의 내용 모델 == “ANY” then return “yes”;
else
   if X == parentX의 마지막 자식 then
      if X == 마지막 상태 then return “no”;

   if previsouX와 똑같은 이름을 가지는 엘리먼트가 두 개 이상 then
      previousX := parentX의 첫 번째 자식;  X := previousX의 next sibling;
      while parentX의 모든 자식 do
         if deleteValidationProcess(previousX, X, nextX) == false then return “no”;
         previousX := X;  X := X의 next sibling;
      end
      return “yes”;
   end if
   else return deleteValidationProcess(previousX, X, nextX);
end else
else return “no”;
```

Algorithm2 : deleteValidationProcess (삭제시 DFA에서 전이 가능 검사)
Input : 이전 엘리먼트 *previousX*, 엘리먼트 *X*, 다음 엘리먼트 *nextX*
Output : 전이가 존재하면 “yes” 아니면 “no” 리턴
Steps :

```
if previousX 다음에 nextX로 전이 가능 then return “yes”;
else return “no”;
```

그림 9 엘리먼트 삭제 검증에 대한 알고리즘

먼트인 G가 전이할 수 없다.

이 삽입 연산은 조건 (마)를 만족시키지 못하므로 유효하지 않다.

● 엘리먼트 삭제

엘리먼트가 삭제될 때의 검증 과정은 그림 9의 알고리즘을 따른다.

엘리먼트를 삭제하려면 우선 부모 엘리먼트가 "ANY"로 선언되어 있는지, 마지막 자식이면 마지막 상태인지 검사해야 한다. 부모 엘리먼트가 "ANY"로 선언되어 있으면 어떤 엘리먼트가 자식 엘리먼트가 존재해야 하기 때문에 이 엘리먼트를 삭제할 수 없다.

이러한 과정을 거친 후 삭제하는 엘리먼트의 이전 엘리먼트 뒤에 삭제할 엘리먼트 다음 엘리먼트가 나올 수 있는지 검사한다. 이것을 검증하기 위해서는 우선 삭제할 엘리먼트의 이전 엘리먼트를 찾고 이 엘리먼트 다음에 삭제할 엘리먼트의 다음 엘리먼트가 나올 수 있는지 검사해야 한다.

또한 삭제의 경우도 삽입의 경우와 마찬가지로 만약 하나의 엘리먼트 선언에서 삭제하는 엘리먼트의 이전 엘리먼트와 똑같은 이름의 엘리먼트가 2개 이상 선언되어 있으면 이 엘리먼트를 찾기가 힘들어진다. 따라서 이 경우에는 자식들 모두를 처음부터 검사해야 한다.

다음은 하나의 엘리먼트가 삭제되었을 때 위의 조건에 따라 검증하는 과정을 예로 보인 것이다.

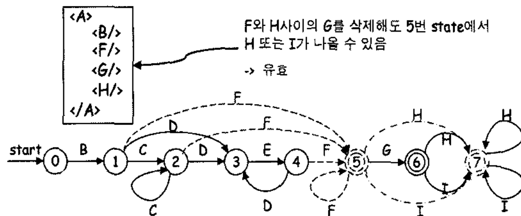

그림 10 엘리먼트 삭제 연산에 대한 검증

(가) 부모 엘리먼트인 A는 "ANY"로 선언 되어 있지 않다.

(나) 삭제하려는 엘리먼트 G는 마지막 엘리먼트가 아니다.

(다) 삭제할 곳 바로 전 엘리먼트의 도착 상태가 5번 하나이다.

(라) 엘리먼트인 H나 I가 상태 5번에서 상태 7번으로 전이할 수 있다.

순서도에서의 조건이 모두 만족하므로 이 삭제 연산은 유효한 연산이 된다.

● 엘리먼트 변경

엘리먼트의 변경은 우선 변경될 엘리먼트를 삭제한 후에 새로운 엘리먼트를 삽입하는 과정으로 이루어질 수 있다. 따라서 엘리먼트를 삭제와 삽입을 동시에 수행하면 된다.

3.3.3 애트리뷰트 검증 과정

● 엘리먼트 삽입

엘리먼트를 삽입할 때 엘리먼트만 삽입될 수 있지만 엘리먼트에 애트리뷰트도 포함될 수 있다. 따라서 애트리뷰트를 가진 엘리먼트를 삽입할 경우 애트리뷰트가 "REQUIRED"로 선언되어 있는지 조사해야 한다. 만약 "REQUIRED"로 선언되어 있는데 삽입하는 엘리먼트가 애트리뷰트를 가지고 있지 않으면 이 연산은 유효하지 않게 된다.

● 엘리먼트 삭제

엘리먼트를 삭제할 경우 만약 ID를 가지고 있으면 해당 ID에 대한 참조가 있는지 검사한다. 만약 ID에 대한 참조가 존재하면 삭제할 수 없다.

● 애트리뷰트 삽입/변경

(가) 애트리뷰트가 DTD에 정의되어 있는가.

(나) 애트리뷰트가 XML문서에 이미 존재하는가.

- 애트리뷰트가 이미 있으면 변경하고 없으면 삽입한다.

(다) 디폴트 타입 검사

- "FIXED"로 선언되어 있으면 애트리뷰트값은 DTD에서 선언된 값과 일치해야 한다.
- "REQUIRED"로 선언되어 있고 애트리뷰트 값이 NULL로 삽입/변경된다면 유효하지 않은 연산이 된다.
- "DEFAULT"로 선언되어 있고 애트리뷰트 값이 NULL로 삽입/변경되면 DTD에서 선언된 값으로 바뀌어야 한다.

(라) 애트리뷰트 타입 검사

- 바꾸는 값이 ID, NMTOKEN, NOTATION, ENUMERATION등과 같은 애트리뷰트 타입에 맞는지 검사한다.
- 만약 삽입하려는 애트리뷰트 타입이 IDREF이면 이 IDREF에 대한 ID가 존재하는지 검사한다.

● 애트리뷰트 삭제

(가) 디폴트 타입 검사

- "FIXED"로 선언되어 있으면 삭제할 필요가 없으므로 삭제하지 않는다.
- "REQUIRED"로 선언되어 있으면 유효하지 않은 연산이 된다.
- "DEFAULT"로 선언되어 있으면 DTD에서 선언된 디폴트 값으로 바뀌어야 한다.

(나) 애트리뷰트 타입 검사

- 만약 삭제하려는 애트리뷰트 타입이 ID이면 이 ID에 대한 IDREF가 존재하는지 검사한다.

4. 성능 평가

이 장에서는 지금까지 구현된 XML 즉시 부분 검증기의 성능을 평가한다. 현재 본 연구와 같이 즉시 부분 검증 방법을 구현한 시스템이 없기 때문에 다른 시스템과의 직접적인 비교는 힘들다. 따라서 우선 본 시스템에서 변경 연산을 수행할 때의 검증 성능을 평가 분석하고, 현재 대부분의 XML저장관리 시스템이 사용하고 있는 지연 전체 검증 방법과 본 시스템의 즉시 부분 검증 방법의 검증 성능을 비교한다. 아래에서 사용되는 모든 성능 분석은 변경 연산 수행시의 성능을 고려한다. 본 논문에서 제안한 즉시 부분 검증 기법을 위해서는 DTD를 처음 저장할 때 DTD정보를 저장해야 하는데 한 번 저장된 후에는 이 정보에 대한 변경이 필요하지 않기 때문에 여기에서는 이에 대한 성능은 고려하지 않는다.

4.1 성능 실험

이 연구에서 구현한 즉시 부분 검증 모듈은 C++로 구현되었으며, Ultrasparc 5500, 메모리 1GB의 하드웨어와 Solaris2.7의 운영체제를 가지는 중형컴퓨터에서 테스트되었다. 그리고 데이타베이스는 Oracle8.1.7을 사용하였다.

표 2는 표 1에 나오는 변경 연산의 종류별로 변경을 수행하였을 때 소요된 검증 시간을 측정한 것이다. 표 1의 변경 연산은 크게 엘리먼트 삽입, 엘리먼트 삭제, 애트리뷰트 삽입/변경, 애트리뷰트 삭제 이렇게 4가지 연산으로 나눌 수 있기 때문에 이 네 가지에 대한 시간을 측정하였으며, 테스트 데이타는 Jon Bosak이 shakespeare희곡을 XML파일로 만든 것들 중에서 hamlet.xml 파일을 사용하였다. hamlet.xml은 280KB의 크기에 19841개의 엘리먼트를 가지고 있지만 애트리뷰트가 없기 때문에 ACT엘리먼트와 SCENE 엘리먼트에 임의로 애트리뷰트를 하나씩 추가할 수 있도록 DTD를 수정하였다.

이 실험에서 우선 하나의 노드를 처리하는 변경연산에 대해 성능을 측정하였고 엘리먼트를 변경할 때 서브트리를 삽입 또는 삭제할 수 있기 때문에 서브트리가 가지고 있는 엘리먼트의 개수를 증가시키면서 테스트하였다. 이 경우 엘리먼트 삽입 위치는 두 개의 형제 노드 사이에 삽입하는 것으로 하였다.

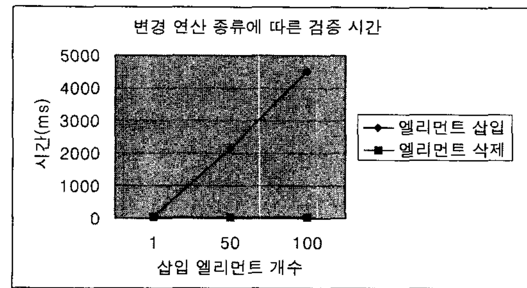

그림 11 변경 연산 종류에 따른 검증 시간

테스트 결과 본 시스템에서는 엘리먼트 삽입의 경우 엘리먼트를 삽입할 때 서브트리를 순회하면서 하나씩 검증하기 때문에 서브 트리에 존재하는 엘리먼트의 개수에 비례하는 것을 볼 수 있다. 하지만 엘리먼트 삭제의 경우 서브 트리의 맨 상위 노드만 검증하기 때문에 엘리먼트 개수에 상관없이 일정한 성능을 보인다. 엘리먼트를 삭제할 때 서브트리 안에 ID가 존재하는지 검사하지만 한번에 검사하기 때문에 성능에 큰 영향이 없으며, 하나의 엘리먼트를 삭제할 경우에는 하나의 엘리먼트를 삽입하는 경우보다 DFA에서 검증하는 엘리먼트 수가 적기 때문에 삽입 시간보다 적게 걸린다.

애트리뷰트의 경우 DOM Level 1에서는 하나의 변경 연산에서 여러 개의 애트리뷰트를 변경할 수 없기 때문에 하나의 애트리뷰트를 변경하는 경우만 고려하였다.

만약 변경하는 노드의 이전 노드와 똑같은 이름을 가지면서 서로 인접하지 않는 형제 엘리먼트 개수가 두 개 이상일 경우 하나의 엘리먼트 삽입/삭제하는 연산을 검증하기 위해서 모든 자식 노드들을 검사해야 하는데 이 경우 모든 자식 노드들을 검사하기 때문에 삽입 연산과 삭제 연산의 시간은 차이가 나지 않는다. 따라서 이 때의 검증시간은 형제 엘리먼트 개수에 비례한다.

4.2 부분 검증과 전체 검증 방법의 성능 비교

이 절에서는 이 연구에서 구현한 부분 검증 방법과 대부분의 XML데이타베이스 시스템이 사용하고 있는 전체 검증 방법의 성능을 이론적으로 비교한다. 다음은 성능 분석시 사용하는 파라미터들이다.

- T_f : 전체 검증 방법의 소요 시간
- T_p : 부분 검증 방법의 소요 시간

표 2 표 1의 변경 연산 종류에 따른 검증 시간

변경연산종류 \ 삽입 엘리먼트 개수	1	10	50	100
엘리먼트 삽입	44ms	457ms	2152ms	4511ms
엘리먼트 삭제	29ms	28ms	31ms	30ms
애트리뷰트 삽입/변경	22ms	-	-	-
애트리뷰트 삭제	19ms	-	-	-

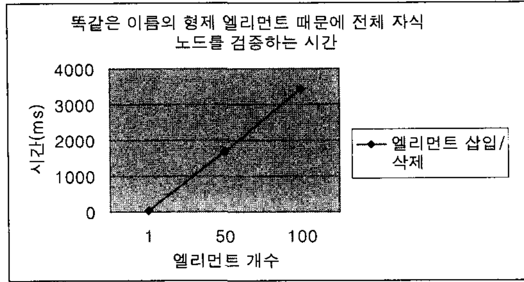

그림 12 똑같은 이름의 형제 엘리먼트 때문에 전체 자식 노드를 검증하는 시간

- T_{rd} : 데이타베이스에서 하나의 DTD를 읽는 시간
- T_{rt} : DFA에서 하나의 엘리먼트를 읽는 시간
- T_{re} : 데이타베이스에서 하나의 엘리먼트를 읽는 시간
- N_e : XML문서에 포함된 엘리먼트 개수
- T_v : 하나의 엘리먼트를 검증하는 시간
- N_b : 형제 엘리먼트 개수
- N_u : 변경하는 노드의 이전 노드와 똑같은 이름을 가지면서 서로 인접하지 않는 형제 엘리먼트 개수

◎ XML 문서 전체 검증 방법의 성능

변경 연산 수행시 전체 검증을 하려면 우선 데이타베이스에 저장된 DTD와 XML문서를 읽어 온 후에 이것을 다시 파서로 파싱하는 과정을 거쳐야 한다. 따라서 전체 검증 시간은 데이타베이스에서 DTD와 XML문서를 읽는 시간에 이것을 파서에 의해서 검증하는 시간이 더해진다. 이를 식으로 표현하면 다음과 같다.

- T_f = DTD 읽는 시간 + XML문서 읽는 시간 + 검증 시간
- 검증 시간 = 하나의 엘리먼트 검증 시간 * 엘리먼트 개수
- $T_f = T_{rd} + T_{re}*N_e + (T_v*N_e) = T_{rd} + (T_{re}+T_v)*N_e$

◎ 부분 검증 방법의 성능

부분 검증 방법은 변경된 부분만 검증하기 때문에 전체 검증 방법과는 달리 데이타베이스에 저장된 DTD와 XML문서를 읽을 필요가 없다. 또한 부분 검증에서는 검증해야 하는 노드는 변경하는 노드, 변경하는 노드의 이전 노드, 변경하는 노드의 다음 노드 이렇게 3개 가 된다. 하지만 만약 변경하는 노드의 이전 노드와 똑같은 이름을 가지면서 서로 인접하지 않는 형제 엘리먼트 개수가 두 개 이상일 경우 즉, $N_u > 1$일 경우 모든 자식 노드를 검증해야 하는데 이 경우 N_b배의 시간이 더 소요된다. 이를 식으로 표현하면 다음과 같다.

- Nu == 1인 경우
  - T_p = DFA에서 세 개의 엘리먼트를 읽는 시간 + XML문서에서 세 개의 엘리먼트를 읽는 시간 + 검증 시간
  - 검증 시간 : 1개 엘리먼트 검증 시간 * (변경하는 노드의 이전 노드 + 변경할 노드 + 변경하는 노드의 다음 노드)
  - $T_p = T_{rt}*3 + T_{re}*3 + T_v*3 = 3*(T_{rt}+T_{re}+T_v)$
- $N_u > 1$인 경우
  - T_p = DFA에서 N_b 개의 엘리먼트를 읽는 시간 + XML문서에서 N_b 개의 엘리먼트를 읽는 시간 + 검증 시간
  - 검증 시간 : 1개 엘리먼트 검증 시간 * N_b
  - $T_p = T_{rt}*N_b + T_{re}*N_b + T_v*N_b = N_b*(T_{rt} + T_{re} + T_v)$

◎ 성능 분석

XML문서가 변경되었을 때 이를 검증하는 시간에 가장 큰 영향을 미치는 것은 XML문서를 구성하는 엘리먼트 개수이다. 검증해야 하는 엘리먼트 개수가 많을수록 시간이 많이 소요되기 때문이다. 따라서 위의 두 방법에서 측정된 시간을 엘리먼트의 개수에 따라 분석해 볼 수 있다.

그림 13은 N_u가 1이고 변경하는 엘리먼트의 개수가 1개일 때 XML문서를 구성하는 엘리먼트의 개수를 증가시키면서 전체검증방법과 부분검증방법의 성능을 그래프로 나타낸 것으로 파라미터의 값을 T_{rd}=10ms,

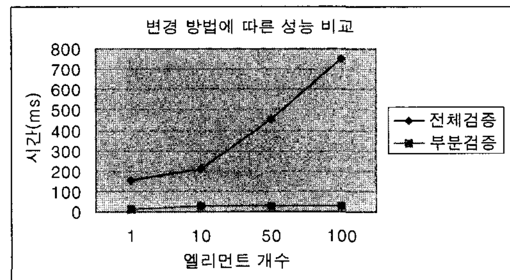

그림 13 변경 방법에 따른 성능 비교

표 3 똑같은 이름의 형제 엘리먼트 때문에 전체 자식 노드를 검증하는 시간

형제 엘리먼트 개수 / 변경연산종류	1	10	50	100
엘리먼트 삽입/삭제	35ms	347ms	1691ms	3428ms

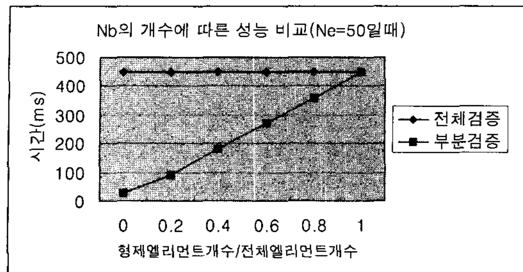

그림 14 N_b의 개수에 따른 성능 비교(N_e=50일 때)

T_{rt}=3ms, T_{re}=4ms, T_v=2ms로 가정하였다. 그래프에서 보면 XML문서를 구성하는 엘리먼트의 개수가 많으면 많을수록 전체검증의 성능은 무한대로 증가하게 되고, 부분검증의 성능은 엘리먼트의 개수에 상관없이 일정하다. 따라서 N_u가 1일 경우 엘리먼트 개수에 상관없이 부분검증의 성능은 항상 전체검증 성능보다 좋은 것을 알 수 있다.

그림 14는 N_u가 1보다 클 경우 형제엘리먼트 개수가 전체엘리먼트 개수에서 차지하는 비율에 따라 전체검증방법과 부분검증방법의 성능을 그래프로 나타낸 것이다. 이 때 N_e는 100으로 고정시켰으며 위의 실험과 같이 변경하는 엘리먼트의 개수는 1개이고 나머지 파라미터도 동일하게 하였다.

그림을 보면 형제엘리먼트개수/전체엘리먼트개수 비율이 증가할수록 부분검증 시간이 증가하는 것을 볼 수 있으며, 최악의 경우 이 비율의 값이 1이 되면 전체 검증과 부분 검증의 값이 같아진다. 이 경우 전체검증과 부분검증 모두 검증하는 엘리먼트 수가 같아지기 때문이다. 하지만 부분검증방법이 전체검증 방법보다 나쁜 성능을 보이지는 않는다.

이 절에서는 변경 연산 수행시 전체를 검증하는 방법과 부분만 검증하는 방법에 대한 성능 평가를 하였다. 성능 평가에 따르면 부분검증을 사용하는 것이 전체검증을 사용하는 것보다 항상 좋은 것을 알 수 있으며, 부분검증시 변경하는 노드의 이전 노드와 똑같은 이름을 가지면서 서로 인접하지 않는 형제 엘리먼트가 많을수록 시간이 오래 걸리지만 전체검증방법의 성능보다 나쁘지 않은 것을 알 수 있다.

5. 결 론

본 논문에서는 데이타베이스에 저장된 XML문서를 변경할 때 변경된 부분만 즉시 검증할 수 있는 메카니즘을 제안하였다. 이 방법에서는 DTD정보를 추출할 때 기존의 XML파서를 수정하여 사용하였으며 사용자가 XML문서를 변경할 때 이 연산이 유효한지 XML 데이타베이스 시스템이 먼저 검사하고 유효하면 연산을 수행하도록 한다.

따라서 XML 데이타베이스 시스템에서 이 방법을 사용하면 항상 유효한 XML문서를 유지할 수 있을 뿐만 아니라 최근에 변경 연산에 대한 연구들에서 발생한 충돌 문제를 해결할 수 있다.

또한 이 방법은 다른 XML데이타베이스 시스템이 변경된 문서를 검증할 때 문서 전체를 검증하는데 반해 변경된 엘리먼트, 변경할 엘리먼트의 이전 엘리먼트, 변경할 엘리먼트의 이후 엘리먼트 이렇게 3개의 엘리먼트를 검증한다. 이렇게 변경된 부분만 검증하기 때문에 XML문서의 엘리먼트 개수에 상관없이 빠른 검증과 변경 연산을 수행할 수 있다.

이 메카니즘은 본 연구실에서 구현한 XD^2M^2(XML Data Document Management Middleware)[15-18]라는 시스템 안에 변경/검증 관리기라는 모듈로 구현되었다. XD^2M^2는 XML 데이타 또는 문서를 RDBMS에 저장, 질의하고 관리하는 시스템으로 W3C의 DOM Level 1과 XQL[19]을 지원한다. 또한 데이타베이스에 저장된 XML문서에 대해 검색과 변경을 동시에 효율적으로 지원할 수 있는 구조로 설계되어 있어 본 연구의 메카니즘을 적용하기에 적당한 시스템이다.

앞으로 위와 같은 XML 저장, 검색 시스템뿐만 아니라 전자 상거래나 정보 통합 시스템등에 이 기법을 적용한다면 다양한 사용자들의 검색, 변경 요구를 충족시킬 수 있을 것이다.

참 고 문 헌

[1] Igor Tatarinov, Zachary G. Ives, Alon Y. Halevy, Daniel S. Weld, *"Updating XML,"* In Proc. of ACM SIGMOD Conference, pp.413-424, 2001.

[2] Jonathan Robie, Ratrick Lehti, *"Updates in XQuery,"* In Proc. of XML Conference, 2001.

[3] Software AG, *QuiP: a prototype of XQuery*, http://www.softwareag.com/developer/quip/default.htm

[4] eXcelon corp., *eXcelon User Guide*: Updating XML data, 1998.

[5] Dr.Harald Schoning, *"Tamino-a DBMS Designed for XML,"* In Proc. of International Conference on Database Engineering, pp.149-154, 2001.

[6] Michael Rys, *"State-of-the-Art XML Support in RDBMS: Microsoft SQL Server's XML Features,"* IEEE Data Engineering Bulletin, Vol.24, No.2, pp.3-11, 2001.

[7] S.Banerjee, V.Krishnamurthy, M.Krishnaprasad, R.Murthy, *"Oracle 8i-the XML Enabled Data Management System,"* In Proc. of International Conference on Database Engineering, pp.561-568, 2000.

[8] J.Cheng, J.Xu, *"XML and DB2,"* In Proc. of International Conference on Database Engineering,

pp.569-573, 2000.

[9] M.Fernandez, W.Tan, D.Suciu, *"Publishing Relational Data in XML:the SilkRoute Approach,"* IEEE Data Engineering Bulletin, Vol.24, No.2, pp.12-19, 2001.

[10] J.Shanmugasundaram, J.Kieman, E.Shekita, C.Fan, J.Funderburk, *"Querying XML Views of Relational Data,"* In Proc. of VLDB Conference, pp.261-270, 2001.

[11] C.M.Sperberg-McQueen, *Notes on schema-validation results*, In http://www.w3.org/ People/ cmsmcq/2001/validation-results, 2001.

[12] W3C working draft. *XQuery: An XML query language*, In http://www.w3.org/TR/xquery, Dec. 2001.

[13] W3C Recommendation. *DOM(Document Object Model) Level 1*, Inhttp://www.w3.org/TR/REC-DOM-Level-1.

[14] Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman, *Compilers Principles, Techniques, and Tools*, Addison-Wesley, 1986.

[15] 김상균, 전희영, 이명철, 이경하, 이규철, 이미영, 손덕주, "XML데이타베이스의 능동적 검증 기법", 27th 한국정보과학회(KISS) 가을 학술발표논문집(I), pp.12 14, 2000.

[16] 연제원, 조정수, 이강찬, 이규철, "XML 문서 구조검색을 위한 저장 시스템 설계", 26th 한국정보과학회(KISS) 봄 학술발표논문집(B), pp.3-5, 1999.

[17] 연제원, 이강찬, 이규철, 나중찬, 이미영, "효율적 XML 문서 변경 및 검색을 위한 페이징 기법", 26th 한국정보과학회(KISS) 가을 학술발표논문집(I), pp.99-101, 1999.

[18] 이명철, 김상균, 이규철, 손덕주, 김명준, "효율적 구조 질의를 지원하는 바다-IV/XML 질의처리기의 설계 및 구현", 정보기술과 데이타베이스 저널 Vol.7, No.2, pp.17-32, 2000.

[19] J. Robie, J. Lapp, and D. Schach, *XQL(XML Query Language)*, The position paper in http:// www.w3.org/TandS/QL/QL98/pp/xql.html

김 상 균

1999년 충남대학교 공과대학 정보통신공학과(공학사). 2001년 충남대학교 공과대학 컴퓨터공학과(공학석사). 2001년~현재 충남대학교 공과대학 컴퓨터공학과 박사과정. 관심분야는 데이타베이스, XML, XML Validation, 전자 상거래

이 규 철

1984년 서울대학교 공과대학 컴퓨터공학과(공학사). 1986년 서울대학교 공과대학 컴퓨터공학과(공학석사). 1990년 서울대학교 공과대학 컴퓨터공학과(공학박사) 1994년 미국 IBM Almaden Research Center 초빙 연구원. 1995년~1996년 미국 Syracuse University, CASE Center 초빙 교수. 1997년~1998년 학술진흥재단 부설 첨단학술정보센터 파견교수 1999년~현재 한국정보과학회 논문편집위원. 2000년~현재 한국 ebXML 전문위원회 위원장. 2001년~현재 전자상거래 표준화 통합 포럼 전자거래 기반 기술위원회 부위원장 2003년~현재 한국전자거래학회 편집이사. 현재 충남대학교 공과대학 컴퓨터공학과 교수. 관심분야는 데이타베이스, XML, 정보 통합, 전자 상거래