XML 뷰 인덱싱
(XML View Indexing)

김 영 성 † 강 현 철 ††
(Young-Sung Kim) (Hyunchul Kang)

요 약 뷰는 이질적인 데이타의 통합 및 여과(filtering) 기능을 통해서 데이타베이스의 필요한 부분을 제공한다. 많은 정보가 쏟아지고 있는 웹 환경의 데이타 교환 표준인 XML에 대해서도 뷰의 개념은 유용하다. 본 논문은 XML 뷰 인덱싱이라고 명명한 XML 뷰를 구현하는 기법을 제안한다. XML 뷰는 XML 뷰에 대한 정보와 더불어 뷰를 구성하는 하부 XML 엘리먼트에 대한 식별자를 저장하는 구조인 XML 뷰 인덱스로 표현된다. XML 뷰 인덱스가 XML 엘리먼트 자체가 아닌 식별자만을 저장하므로, 사용자가 XML 뷰를 요청하면 하부 XML 문서를 기반으로 XML 뷰를 실체화해야 한다. 또한, 하부 XML 문서에 대한 변경에 대하여 XML 뷰 인덱스의 일관성을 유지하기 위한 효율적인 점진적 갱신 기법이 필요하다. 본 논문에서는 XML 뷰 인덱싱을 위한 자료구조와 알고리즘을 제안하고 구현하였다. 성능 평가 결과 XML 뷰 인덱싱을 사용하는 것이 매번 뷰를 재생성하는 경우보다 질의 재수행 시간이 적게 걸렸다. XML 뷰 인덱싱 기법이 실체화 시간으로 인해 XML 실체뷰 기법보다 질의 재수행 시간은 많이 걸리지만, 저장 공간 면에서는 약 30배 정도 효율적인 것으로 나타났다.

키워드 : XML, XML 뷰, XML 뷰 인덱싱, 뷰 실체화, 점진적 뷰 갱신

Abstract The view mechanism provides users with appropriate portions of database through data filtering and integration. In the Web era where information proliferates, the view concept is also useful for XML, a future standard for data exchange on the Web. This paper proposes a method of implementing XML views called XML view indexing, whereby XML view xv is represented as an XML view index(XVI) which is a structure containing the identifiers of xv's underlying XML elements as well as the information on xv. Since XVI for xv stores just the identifiers of the XML elements but not the elements themselves, when a user requests to retrieve xv, its XVI should be materialized against xv's underlying XML documents. Also an efficient algorithm to incrementally maintain consistency of XVI given a update of xv's underlying XML documents is required. This paper proposes and implements data structures and algorithms for XML view indexing. The performance experiments on XML view indexing reveal that it outperforms view recomputation for repeated accesses to the view, and requires as much as about 30 times less storage space compared to XML view materialization though the latter takes less time for repeated accesses to the view due to no need of materialization.

Key words : XML, XML view, XML view indexing, view materialization, incremental view refresh

1. 서 론

데이타베이스 시스템에서 뷰는 데이타를 접근하고 제어하는 유용하고 효과적인 메카니즘이다. 뷰는 데이타 관리와 데이타베이스 설계의 많은 면과 관련이 있다. 뷰의 가장 중요한 응용 중 하나는 정보 필터링과 통합이다. 이것은 오늘날 이질적인 많은 정보가 매일 쏟아지는 웹 기반 환경에서 정보 처리를 위한 중요한 기능이다.

웹 상의 데이타 교환 표준으로 XML이 출현한 이래, XML 데이타 관리에 대한 많은 이슈들이 연구되어 왔다. 뷰는 XML 문서에 대해서도 유용한 개념이다[1, 2, 3, 4]. XML 문서에 대해 자주 제기되는 질의에 대해서 편리하고 효과적인 재사용을 위해 XML 뷰를 정의할 수 있다.

XML 뷰가 효과적으로 이용될 수 있는 응용 중 하나

· 이 논문은 2002학년도 중앙대학교 학술연구비 지원에 의한 것임.
† 비 회 원 : 중앙대학교 컴퓨터공학과
yskim@dblab.cse.cau.ac.kr
†† 종신회원 : 중앙대학교 컴퓨터공학과 교수
hckang@cau.ac.kr
논문접수 : 2002년 6월 21일
심사완료 : 2003년 2월 12일

는 XML 웨어하우스이다. 웨어하우스에 있는 방대한 양의 XML 문서 중 자주 접근되는 부분을 뷰로 정의할 수 있으며, 질의 성능을 위하여 실체화하여 유지할 수 있다. 최근에는 질의 처리, 변경 제어, 데이타 통합과 같은 데이타베이스 형태의 서비스를 제공하는 Xyleme 프로젝트가 수행되었고[5] 이를 통해 설립된 Xyleme사는 웹 상에 존재하는 XML 문서로 구성된 동적이고 방대한 웨어하우스를 상업적으로 서비스하기 위한 제품을 개발 중이다[6].

XML 뷰가 효과적으로 이용될 수 있는 또 다른 웹 응용은 전자상거래이다. 현재 전자상거래를 위한 응용 프로그램은 XML 데이타를 요구하고 있지만, 대부분의 비즈니스 데이타는 XML 형태가 아닌 관계 데이타베이스에 저장되어 있다. 따라서 이러한 관계 데이타는 종종 XML 형태로 출판될 필요가 있다[7,8]. 이를 위해 자주 요구되는 XML 출판 결과를 뷰로 정의하여 사용할 수 있다.

사용자가 같은 질의를 반복적으로 제기할 필요 없이 항상 최신의 결과를 얻을 수 있도록 하는 기능이 현재 인터넷과 같은 환경에서 널리 요구되고 있다[9]. XML 뷰는 웹 상에 존재하는 방대한 XML 문서에 대하여 이와 같은 연속적 질의(continuous query) 능력을 제공하는 데에도 효과적으로 이용될 수 있다.

본 논문은 XML 뷰 인덱싱이라고 명명한 XML 뷰를 구현하는 방법을 제안한다. 본 논문의 XML 뷰는 XML 웨어하우스에 있는 하부 XML 문서에 대한 XML 질의 표현으로부터 추출된 내용으로서 XML 형태이며, 질의의 조건을 만족하는 XML 엘리먼트로 구성된다. 제안하는 XML 뷰 인덱싱에서 XML 뷰는 XML 뷰 인덱스로 표현된다. XML 뷰 인덱스는 XML 뷰에 대한 정보와 더불어 XML 뷰를 구성하는 엘리먼트 식별자를 포함하는 구조이다. XML 뷰에 대한 XML 뷰 인덱스가 엘리먼트 자체가 아니라 엘리먼트 식별자만 저장하기 때문에 사용자가 XML 뷰를 추출하고자 하는 경우, XML 뷰 인덱스는 XML 뷰의 하부 XML 문서를 대상으로 실체화되어야 한다.

XML 뷰를 접근할 때 높은 성능을 얻기 위하여 XML 뷰를 구현하는 다른 방법은 XML 뷰를 실체화된 형태로 유지하는 것이다. 실체뷰와 뷰 인덱스는 모두 하부 XML 문서가 변경되었을 경우 일관성을 유지하기 위한 오버헤드를 수반한다. 두 방법 모두 XML 뷰의 빠른 검색을 가능하게 하지만 일관성 유지를 위한 오버헤드를 고려하지 않았을 때 실체뷰가 뷰 인덱싱보다 뷰 검색 시간은 더 빠르다. 하지만, 실체뷰 기법은 많은 수의 XML 뷰를 유지해야 할 경우 심각한 공간 오버헤드를 야기한다. XML 뷰 인덱싱은 XML 뷰 인덱스를 실체화하기 위한 추가적인 시간을 필요로 하지만, 많은 수의 XML 뷰를 지원할 때 공간 문제를 해결할 수 있다.

본 논문은 XML 뷰 인덱싱 기법에서의 자료 구조와 알고리즘을 제안하고, 구현 및 성능 평가 결과를 기술한다. XML 뷰 인덱싱에 대한 주요 연구 이슈로는 첫째, XML 뷰 인덱스 및 그것의 점진적 갱신을 위한 XML 뷰 인덱스의 자료구조, 둘째, XML 뷰의 정의에 따른 XML 뷰 인덱스 생성 알고리즘, 셋째, 뷰 인덱스로부터 효율적으로 하부 XML 문서를 접근하여 뷰를 구성하는 실체화 알고리즘, 넷째, 하부 XML 데이타의 변경 내용이 관련 뷰에 영향을 미치는 지의 여부를 점검하는 연관성 검사 기법, 그리고 다섯째, XML 뷰 인덱스의 점진적 갱신 알고리즘 등을 들 수 있다. XML 뷰 인덱스의 자료 구조는 이러한 알고리즘들을 효율적으로 지원할 수 있도록 설계되어야 한다.

본 논문의 구성은 다음과 같다. 2장에서는 관련연구를 기술한다. 3장에서는 XML 뷰 관리 시스템, 하부 XML 데이타 저장 구조, XML 질의어, XML 뷰 정의, 변경 모델 등을 기술하고, 4장에서는 XML 뷰 인덱스의 구조를 제안하고 뷰 인덱스 생성 알고리즘 및 뷰 인덱스로부터 뷰를 구성하는 뷰 실체화 알고리즘을 기술한다. 5장에서는 하부 XML 데이타의 변경에 따른 연관성 검사 기법과 점진적 갱신 알고리즘을 기술한다. 6장에서는 본 논문에서 제안한 XML 뷰 인덱스 구조 및 이와 관련된 알고리즘을 구현하여 성능 평가를 수행한 결과를 기술한다. 마지막으로 7장에서 결론을 맺고 향후 연구내용을 기술한다.

2. 관련 연구

관계형 데이타베이스에서 뷰를 구현하는 기법으로 처음으로 제안된 기법은 질의 변경 기법[10]이다. 이는 뷰의 정의만을 시스템 목록에 저장해 두었다가 해당 뷰에 대한 질의가 제기되면 뷰의 정의를 바탕으로 하부 데이타베이스에 대한 질의를 생성하여 수행하는 방법이다. 이후에 데이타베이스 스냅샷(snapshot)과 실체뷰 기법이 많은 주목을 받아왔다[11]. 이들 기법에서는 뷰의 내용을 실체화하여 유지하고 이후 하부 데이타베이스의 변경 중 뷰와 관련이 있는 것만 점진적으로 반영한다. 이와 유사한 개념으로 어떤 연산의 결과를 뷰로 정의하고 이를 실체화하기 전단계로서의 뷰 인덱스로 표현하는 방법도 연구되었다[12, 13].

[13]에서는 조인 연산의 결과에 대한 인덱스인 조인 인덱스를 제안하였다. 두 릴레이션 R과 S 간에 조인을 수행하였을 때 그 결과 릴레이션을 구성하는 각 튜플은

R의 한 튜플과 S의 한 튜플로부터 구성된다. 조인 인덱스는 R과 S의 튜플 식별자(surrogate) 쌍의 집합으로 정의된다. 조인 인덱스는 동일한 조인 연산이 다시 요청되었을 때는 물론 세미조인 등 관련된 연산의 수행에 인덱스로서 이용될 수 있다. 또한 하부 데이타 소스인 릴레이션 R과 S가 변경되었을 때 그 내용을 해당 조인 인덱스에 점진적으로 반영하는 알고리즘을 제안하였다. [12]에서는 조인 연산 뿐 아니라 select 연산 결과를 뷰 인덱스로 지원하기 위한 뷰 인덱스의 저장 구조, 점진적 일관성 유지 기법, 그리고 효율적인 실체화 기법을 제안하였다.

한편, XML 데이타와 같은 반구조적(semistructured) 데이타에 대한 실체뷰의 점진적 갱신 기법도 연구되었다[14,15,16]. [15]에서는 에지(edge)가 레이블을 갖는 루트가 있는 트리 구조[17]에 기반을 둔 반구조적 데이타베이스에 대한 실체뷰의 점진적 갱신에 대해 연구하였다. 뷰는 UnQL[17]로 정의한 것으로 조인이 없는 질의만을 대상으로 하였다. 하부 데이타베이스의 변경으로는 한 트리를 기존 노드의 서브트리로 삽입하는 것과 기존 서브트리를 새 트리로 대체(replace)하는 것을 고려하였다. 데이타 소스에 변경이 발생하면 그 소스로부터 생성된 뷰가 있는 사이트로 통지되고, 변경내용 △(즉, 새로 또는 기존 서브트리를 대체하기 위해 삽입된 서브트리)이 전송된다. 실체뷰가 있는 사이트는 전송된 △를 처리하여 실체뷰를 점진적으로 갱신한다. 이 기법은 실체뷰의 갱신을 위해 하부 데이타 소스에의 접근을 피하고 오직 변경내용 △만 접근하면 되지만, 조인 질의에 의한 실체뷰의 점진적 갱신을 지원할 수 없고 하부 데이타 소스의 내용 수정(modification)에 의한 실체뷰의 점진적 갱신을 수행하지 못한다.

[16]에서는 그래프 구조의 데이타베이스를 대상으로 생성된 실체뷰의 점진적 갱신을 연구하였다. 대상 데이타베이스의 예로는 링크로 연결된 웹 페이지(들)을 들 수 있다. 즉, 포인터(에지)를 갖는 객체(노드)들의 집합으로 모델링이 가능한 데이타베이스이다. 뷰는 OQL[18]의 확장 버전으로 정의한 것을 대상으로 하였고, 실체뷰는 뷰의 조건을 만족하는 객체들의 집합으로 표현하고 이들 간의 연결 즉, 에지는 취급하지 않았다. 데이타 소스의 변경으로는 기존의 두 객체 간에 새로운 에지의 삽입, 기존 에지의 삭제, 그리고 원자객체(atomic object) 값의 수정을 고려하였고, 객체의 삽입과 삭제는 대상으로 하지 않았다. 데이타 소스에 변경이 발생하면, 실체뷰에 영향을 미치는 객체들을 알아내기 위한 질의를 생성한 후 이를 데이타 소스에 대해 수행하여 검색된 객체를 실체뷰에/로부터 삽입/삭제한다.

[14]에서는 그래프 기반의 객체 교환 모델(OEM)[19]을 따르는 반구조적 데이타에 대한 실체뷰의 점진적 갱신에 대해 연구하였다. 뷰는 반구조적 데이타베이스 관리 시스템인 Lore[20]의 질의어인 Lorel[21]의 확장 버전을 사용하여 정의된 것을 대상으로 하였고 [16]과 달리 실체뷰에도 객체 간의 에지를 포함시켰다. 고려된 데이타 소스의 변경은 [16]에서와 같은 세가지이고 실체뷰의 점진적 갱신 방식도 동일하다. 즉, 변경 각각에 대해 뷰 유지 문장(view maintenance statement)들을 생성해내고 이들을 하부 데이타 소스에 대해 수행하여 뷰의 갱신에 필요한 객체들을 검색해낸 후 이들을 실체뷰에 반영한다.

이들 연구들은 모두 그래프 기반의 반구조적 데이타로부터 생성된 실체뷰의 점진적 갱신을 다룬 것으로, 고려된 데이타 소스의 변경 유형이 제한적이며, 변경 유형이 아주 제한적인 [15] 외에는 실체뷰의 점진적 갱신을 위해 모든 유형의 변경에 대해 하부 데이타 소스의 검색을 필요로 한다. 또한 가장 복잡도가 높은 실체뷰를 지원하는 [14]의 경우에는 뷰와 연관성이 없는 변경 중 일부에 대해서도 뷰 유지 문장을 생성해야 한다. 이들은 점진적 갱신의 효율성을 저하시키는 요인이 된다.

한편, XML 데이타와 같은 반구조적 데이타에 대한 검색 결과를 뷰 인덱스와 같은 인덱스로 유지하는 기법에 관한 연구는 아직 없고, 주어진 반구조적 질의를 효율적으로 처리하기 위한 전통적인 의미의 인덱싱 기법이 제안되고 있다[22, 23, 24]. 기존의 관계 또는 객체지향 데이타베이스에서의 인덱싱은 고정된 스키마 하에서 데이타 타입을 아는 애트리뷰트 (관계 데이타베이스의 경우) 또는 특정 경로(path) (객체지향 데이타베이스의 경우)에 대해 인덱스를 생성한다. 그러나 반구조적 데이타는 고정적인 스키마가 없고 데이타 항목의 불규칙성, 데이타 타입이나 데이타 구조의 이질성이 존재하기 때문에 인덱스를 생성할 대상이 불분명할 수 있다. 현재까지 제안된 반구조적 데이타에 대한 인덱싱 기법은 경로 표현식에 대한 인덱스를 생성하는데, 단순 경로 질의 밖에 지원하지 못하거나, 일반적이고 복잡한 질의를 지원하기 위해서는 그 생성 시간 오버헤드와 인덱스를 저장하기 위한 공간 오버헤드가 커지는 문제점을 갖고 있다.

3. XML 뷰 관리 시스템

본 장에서는 본 논문에서 고려하는 XML 뷰 관리 시스템에 대해서 설명한다. 먼저, XML 뷰 관리 시스템 구조를 살펴보고, 본 논문에서 대상으로 하는 하부 XML 데이타 소스의 DOM[25] 기반 저장 구조에 대해 설명한다. 그리고, 본 논문에서 사용하는 XQuery[26]를

이용한 XML 뷰 모델을 기술한다. 마지막으로 하부 XML 데이타 소스에서의 변경 모델을 정의한다.

3.1 XML 뷰 관리 시스템 구조

본 논문에서 고려하는 XML 뷰 관리 시스템은 그림 1과 같다. XML 뷰 관리 시스템은 XML 저장 관리 시스템이 관리하는 하부 XML 데이타에 대해 정의된 XML 뷰를 관리한다. 본 논문에서는 그림 1과 같이 두 가지 인터페이스를 가지는 XML 저장 관리 시스템을 가정한다. 하나는 XQuery 질의에 대하여 그 결과를 반환해 주는 것이고, 다른 하나는 특정 엘리먼트 식별자(Element ID)를 받아서 그 내용을 반환해 주는 것이다. 이를 위하여 XML 저장 관리 시스템은 XQuery 질의를 처리할 수 있는 능력을 가지며, 엘리먼트 식별자를 기반으로 원하는 엘리먼트에 직접 접근할 수 있는 DOM 기반 저장 구조를 가진다. 또한, XML 저장 관리 시스템은 하부 XML 데이타에 대한 변경에 대하여 XML 뷰 관리 시스템에게 통보해 주는 기능을 갖는다.

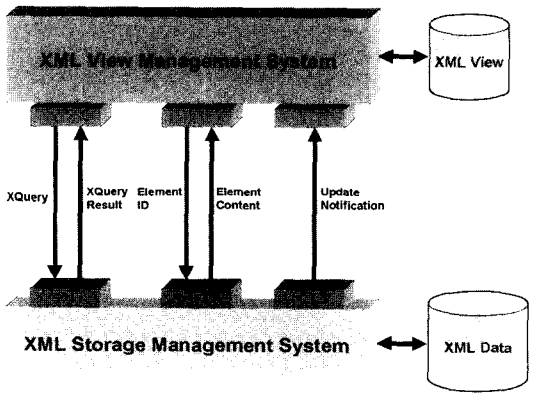

그림 1 XML 뷰 관리 시스템 구조

3.2 하부 XML 데이타 저장 구조

본 논문에서 대상으로 하는 하부 XML 데이타는 DOM 기반 저장 구조에 저장된다. DOM(Document Object Model)은 XML 문서의 내용, 구조, 스타일 등을 접근하고 수정할 수 있도록 해주는 표준 인터페이스이다[25]. DOM은 XML 문서를 표현하는 표준 객체 집합, 표준 데이타 모델인 DOM 구조 모델(DOM Structure Model), 표준 인터페이스 등을 제공한다. DOM 구조 모델은 여러 종류의 노드(Node)들로 구성된다. 본 논문에서는 DOM 구조 모델을 구성하는 노드들 중 XML 질의의 대상이 되는 엘리먼트(Element Node)와 그 값(Text Node) 및 엘리먼트에 속하는 속성(Attribute Node)만으로 이루어진 DOM 구조 모델의 부분 집합만을 고려한다. 그림 2는 본 논문에서 사용하는 XML 데이타의 예를 나타낸 것이다. 그림 2에서 각 엘리먼트의 EID 속성 및 그 값은 실제 원본 XML 데이타에 존재하는 것이 아니라, DOM 기반 XML 저장 관리 시스템에서 할당한 엘리먼트 식별자를 표현한 것이

```
<?xml version="1.0" encoding="euc-kr"?>
<!DOCTYPE USEDCAR SYSTEM "USEDCAR.dtd" [ ]>
<USEDCAR EID="1">
    <DEALER EID="2">
        <NAME EID="3">Lisa Mehler</NAME>
        <EMAIL EID="4">LMehler@aDomain.Com</EMAIL>
        <PHONE EID="5">781 781 790</PHONE>
        <ADDRESS EID="6">
            <STREET EID="7">Central 221</STREET>
            <CITY EID="8">LA</CITY>
            <STATE EID="9">California</STATE>
        </ADDRESS>
        <CAR EID="10">
            <MAKE EID="11">Mercedes Benz</MAKE>
            <MODEL EID="12">M-Class</MODEL>
            <YEAR EID="13">1998</YEAR>
            <PRICE EID="14">29500</PRICE>
            <DRIVE_MILE EID="15">45000</DRIVE_MILE>
            <OPTION EID="16">
                <POWERLOCK EID="17">Yes</POWERLOCK>
                <POWERWINDOW EID="18">Yes</POWERWINDOW>
                <Alarm EID="19">Yes</Alarm>
                <STEREO EID="20">Radio/Cassette/CD</STEREO>
                <Interiors EID="21">Leather</Interiors>
                <AIRCONDITIONER EID="22">Yes</AIRCONDITIONER>
                <STATUS EID="23">Very nice pre-owned</STATUS>
            </OPTION>
        </CAR>
    </DEALER>
    <DEALER EID="24">
        <NAME EID="25">Brian Ricci</NAME>
        <EMAIL EID="26">BRicci@aDomain.Com</EMAIL>
        <PHONE EID="27">781 781 787</PHONE>
        <ADDRESS EID="28">
            <STREET EID="29">Central 221</STREET>
            <CITY EID="30">LA</CITY>
            <STATE EID="31">California</STATE>
        </ADDRESS>
        <CAR EID="32">
            <MAKE EID="33">GMC</MAKE>
            <MODEL EID="34">Jimmy</MODEL>
            <YEAR EID="35">1998</YEAR>
            <PRICE EID="36">31000</PRICE>
            <DRIVE_MILE EID="37">45000</DRIVE_MILE>
            <OPTION EID="38">
                <POWERLOCK EID="39">Yes</POWERLOCK>
                <STEREO EID="40">Radio/Cassette/CD</STEREO>
                <POWERWINDOW EID="41">Yes</POWERWINDOW>
                <Interiors EID="42">Leather</Interiors>
                <AIRCONDITIONER EID="43">Yes</AIRCONDITIONER>
                <STATUS EID="44">Very affordable</STATUS>
            </OPTION>
        </CAR>
    </DEALER>
</USEDCAR>
```

그림 2 XML 데이타 예

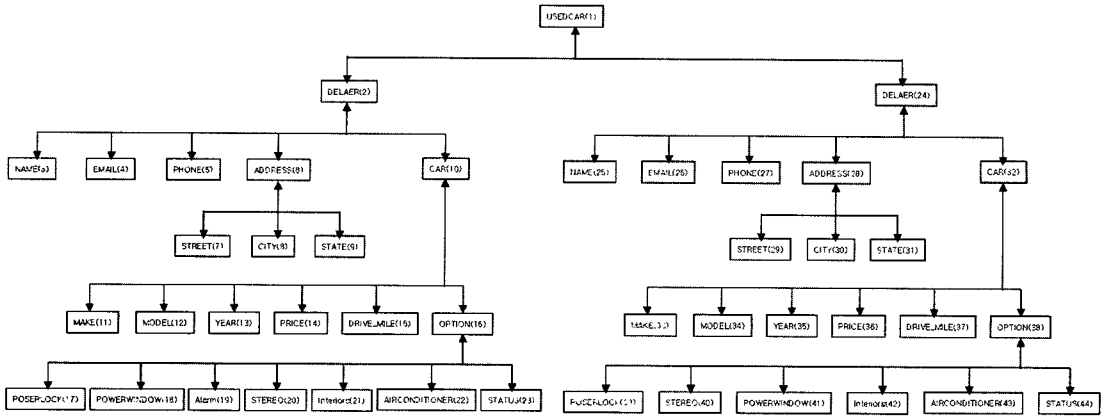

그림 3 그림 2의 XML 데이타에 대한 간략한 DOM 구조 모델

다. 그림 3은 그림 2의 XML 데이타에 대하여 엘리먼트 노드의 부모 자식 관계를 간략한 DOM 구조 모델로 표현한 것이다. 그림 3에서 엘리먼트 이름 다음의 괄호 안의 숫자는 그림 2의 EID 속성 값을 나타낸다. 하부 XML 저장 관리 시스템은 EID 값을 기반으로 해당 엘리먼트에 직접 접근할 수 있는 능력을 지닌다.

3.3 XML 뷰 정의

본 논문에서는 XML 뷰 정의를 위한 XML 질의어로 XQuery를 사용한다. XQuery는 W3C에서 제안한 XML 질의어로서 최근 몇 년간 제안된 여러 XML 질의어 중 현재 표준으로 부각되고 있는 것이다. XQuery는 Quilt[27]라는 XML 질의어를 기반으로 하였으며, XPath[28]와 XQL[29]에서 사용하는 경로 표현(path expression)을 사용한다. 그리고, XML-QL[30]에서와 같이 바인딩 변수(binding variable)를 사용한다.

XML 뷰 정의 언어에서 중점적으로 살펴보아야 하는 특성은 뷰의 결과 형태와 뷰의 조건 형태이다. XML 뷰는 하부 데이타를 기반으로 다른 형태의 XML 데이타를 생성한다. 뷰 결과는 하부 데이타만을 기반으로 생성될 수도 있지만, 하부 데이타 이외에 다른 데이타가 추가되어 생성될 수도 있다. 또한, XML 뷰는 하부 데이타에 대해 직접 정의할 수도 있지만, 이미 존재하는 뷰에 대해 정의하여 사용할 수 있다. 하지만, 이미 존재하는 뷰를 이용하여 정의된 뷰는 순수 하부 데이타 기반의 것으로 변환할 수 있다. 따라서, 본 논문에서는 하부 데이타에 대해 직접 정의하는 XML 뷰만을 대상으로 한다.

본 논문에서 고려한 XQuery를 이용하여 표현된 XML 뷰 정의에 나타나는 조건의 형태 분류는 표 1과 같다. 즉, XML 뷰 정의에 나타나는 각각의 조건은 표 1과 같은 형태 중 하나에 해당되며, 조건절은 각각의 조건을 논리 연산으로 연결한 형태를 지닌다.

또한, 본 논문에서는 XML 뷰 정의에 사용되는 XQuery 질의어의 형식은 하나의 FLWR(FOR, LET, WHERE, RETURN) 문으로 제한한다(단, LET 절은 고려하지 않는다.). 그림 4는 XQuery를 이용하여 그림 2의 XML 데이타에 대하여 XML 뷰를 정의한 것이다. 그림 4의 XML 뷰가 표현하는 질의는 *'LA에 있는 자동차 판매상 중 자동차의 가격이 3만 달러 이하이고, 주행*

표 1 XML 뷰 정의에서의 조건 형태 종류

조건 형태	의미	예
E (Element Existence)	엘리먼트가 존재하는 지에 대한 여부	$P/POWERWINDOW
ER (Element Existence and Relational Operation)	엘리먼트가 존재하고 특정 관계 연산을 만족하는 지에 대한 여부	$C/PRICE <= 30000
NE (Negation of E)	엘리먼트가 존재하지 않는 지에 대한 여부	not $P/AUTOMATIC
NER (Negation of E or Relational Operation)	엘리먼트가 존재하지 않거나 존재할 경우 특정 관계 연산을 만족하는 지에 대한 여부	not $P/Interiors or $P/Interiors = "Leather"

거리가 5만 Km 이하이고, 파워 윈도우이고, 라디오/카세트/CD를 갖춘 오디오를 장착하고, 수동 기어이고, 인테리어 정보가 있다면 가축인 각 자동차의 판매상 이름, 이메일 주소, 전화번호, 자동차 제조사, 모델, 연식, 상태를 반환하라.'라는 것이다. 그림 5는 그림 2의 XML 데이타에 대한 그림 4의 뷰에 대한 결과를 나타낸 것이다.

```
<View_USEDCAR1>

    FOR  $U in //USEDCAR,
         $D in $U/DEALER,
         $A in $D/ADDRESS,
         $C in $D/CAR,
         $P in $C/OPTION
    WHERE
         $A/CITY = "LA" AND
         $C/PRICE <= 30000 AND
         $C/DRIVE_MILE <= 50000 AND
         $P/POWERWINDOW AND
         $P/STEREO = "Radio/Cassette/CD" AND
         not $P/AUTOMATIC AND
         (not $P/Interiors or $P/Interiors = "Leather")
    RETURN
         <CAR>
              $D/NAME
              $D/EMAIL
              $D/PHONE
              $C/MAKE
              $C/MODEL
              $C/YEAR
              $P/STATUS
         </CAR>

</View_USEDCAR1>
```

그림 4 뷰 정의 예

```
<View_USEDCAR1>
  <CAR>
    <NAME EID="3">Lisa Mehler</NAME>
    <EMAIL EID="4">LMehler@aDomain.Com</EMAIL>
    <PHONE EID="5">781 781 790</PHONE>
    <MAKE EID="11">Mercedes Benz</MAKE>
    <MODEL EID="12">M-Class</MODEL>
    <YEAR EID="13">1998</YEAR>
    <STATUS EID="23">Very nice pre-owned  </STATUS>
  </CAR>
</View_USEDCAR1>
```

그림 5 그림 2의 XML 데이타에 대한 그림 4의 뷰 결과

3.4 변경 모델

XML 데이타에 대한 변경은 문서 또는 엘리먼트 단위로 이루어질 수 있다. 문서 단위의 변경에 대하여 점진적으로 뷰를 유지하는 것은 간단하다. 하지만, 엘리먼트 단위의 변경은 복잡한 뷰 유지 기법을 필요로 한다. 본 논문에서는 엘리먼트 단위 변경에 따른 뷰 유지 기법을 다룬다.

표 2 변경 연산에 대한 정보 형태

필드이름	내용
Type	변경 연산 종류 {INSERT, DELETE, CHANGE}
ParentEID	변경 대상 엘리먼트의 부모 엘리먼트 ID
EName	변경 대상 엘리먼트의 엘리먼트 이름
EID	변경 대상 엘리먼트의 엘리먼트 ID
EPath	변경 대상 엘리먼트의 엘리먼트 경로
Value	변경 대상 엘리먼트의 새로운 값 - if Type==DELETE then NULL - if element has no value then NULL

XML 뷰 정의의 조건절 및 선택절에 나타나는 엘리먼트의 추가/삭제/값 변경은 뷰의 결과에 영향을 미친다. 데이타 소스에서 발생할 수 있는 XML 데이타에 대한 엘리먼트 단위의 변경 연산의 종류는 **엘리먼트 추가(INSERT), 엘리먼트 삭제(DELETE), 엘리먼트 값 변경(*CHANGE*)** 등이다. 본 논문에서는 복잡성을 줄이기 위하여 속성에 대한 연산은 다루지 않는다. 본 논문에서는 엘리먼트 단위의 변경이 변경 연산의 종류에 따라 표 2와 같은 형태로 뷰 관리자에게 통보된다고 가정한다.

4. XML 뷰 인덱스

본 장에서는 본 논문에서 제안하는 XML 뷰 인덱스의 구조 및 뷰 인덱스 생성 알고리즘 그리고 뷰 결과 생성을 위한 뷰 *실체화* 알고리즘을 기술한다. 하부 XML 데이타의 엘리먼트 단위 변경에 따른 뷰 인덱스의 점진적 갱신에 대해서는 5장에서 기술한다.

4.1 XML 뷰 인덱스 구조

XML 뷰의 결과는 하부 XML 데이타를 기반으로 한 다른 형태의 XML 데이타로 간주할 수 있다. 기존 데이타와 다른 새로운 구조를 가지긴 하지만 기존 데이타를 바탕으로 한 데이타이기 때문에 이를 효율적으로 저장하고 추출할 수 있는 뷰 인덱스 구조가 필요하다. 뷰 인덱스에는 뷰의 조건을 만족하여 선택되는 엘리먼트에 대한 포인터(식별자)뿐만 아니라 하부 데이타 변경에 따른 점진적 갱신을 위한 부가적인 정보를 저장해야 한다. 이 부가적인 정보에는 하부 데이타에 대한 변경 연산이 뷰 인덱스 구조에 영향을 주는 지의 여부를 판단하는 연관성 검사를 위한 정보가 포함된다.

본 논문에서 제안하는 뷰 인덱스 구조는 트리로서 뷰

표 3 VD 노드 구조 및 정보 (구성에서 * 표시는 0 또는 그 이상 횟수의 반복을 의미)

종 류	구 성
ID : 뷰 식별자	<ID>
View Definition : 뷰 정의	<ViewDefinition>
SEI (Selected Element Information) : 명시적 또는 암시적으로 선택되는 엘리먼트 정보 집합	<SEIndex, BindVar, ParentBindVar, EName, SEP, Type>* - SEIndex : SEI 배열 인덱스 - BindVar : 엘리먼트 바인딩 변수 이름 - ParentBindVar : 부모 엘리먼트 바인딩 변수 이름 - EName : 엘리먼트 이름 - SEP : 엘리먼트 경로 (Selected Element Path) - Type : 엘리먼트 종류 (ISE | ESE)
CEI (Conditional Element Information) : 조건절을 구성하는 각 조건 엘리먼트 정보	<CEIndex, ParentBindVar, EName, CEP, CT, OP, Value>* - CEIndex : CEI 배열 인덱스 - ParentBindVar : 부모 엘리먼트 바인딩 변수 이름 - EName : 엘리먼트 이름 - CEP : 엘리먼트 경로 (Conditional Element Path) - CT : 조건 종류 (E | ER | NE | NER) - OP : CT가 ER 또는 NER인 경우 관계 연산자 종류 - Value : 비교값 (CT가 E 또는 NE인 경우는 NULL)
CES (Condition Evaluation Sequence) : 조건 평가 순서 정보	<CESeq, <<Type, Index>, op, <Type, Index>>>* - CESeq : 조건 평가 순서 번호 - Type : 비교 대상 종류 (CEI | CES) - Index : 비교 대상 배열 인덱스 (CEIndex | CESeq) - op : 논리 연산자
VRS (View Result Structure) : View 결과 생성 정보	<VRIndex, VREPath, <SEIndex>*>* - VRIndex : 뷰 결과 생성 정보 배열 인덱스 - VREPath : 뷰 결과 구성 엘리먼트 경로 - SEIndex : 뷰 결과 구성 엘리먼트의 SEI 배열 인덱스
RootVIE : 최상위 VIE 노드 포인터	<RootVIE>* - RootVIE : 최상위 VIE 노드 포인터

에 대한 메타 정보를 가지는 하나의 **뷰 정의(VD: View Definition) 노드**와 뷰 인덱스를 구성하는 엘리먼트에 대한 정보를 가지는 여러 개의 **뷰 인덱스 엘리먼트(VIE: View Index Element) 노드**로 구성된다. VD 노드는 뷰 정의에 나타나는 정보를 분리하여 저장하기 위한 것이다. VD 노드에 저장되는 정보는 표 3과 같다.

SEI(Selected Element Information)는 VIE 노드를 구성하는 엘리먼트에 대한 정보를 저장한다. VIE 노드를 구성하는 엘리먼트는 XQuery의 선택절(RETURN 절)에 나타나는 뷰 결과를 구성하기 위해 명시적으로 선택되는 엘리먼트(ESE: Explicitly Selected Element) 뿐만 아니라, 연관성 검사 정보를 저장하기 위해 암시적으로 선택되는 엘리먼트(ISE: Implicitly Selected Element)로 구성된다. ISE는 RETURN 절에는 나타나지 않지만, FOR 절에 나타나는 엘리먼트에 해당된다. **CEI(Conditional Element Information)**는 뷰 정의의 조건절(WHERE 절)에 나타나는 정보로서 표 1의 조건 형태에 해당하는 각각의 조건 정보를 저장한다. CEI는 연관성 검사 정보를 생성하는 데 사용된다. **CES(Condition Evaluation Sequence)**는 CEI의 각 조건을 연결한 논리 연산을 평가하는 순서를 나타낸 것이다. **VRS(View**

표 4 그림 2/그림 4의 뷰 정의에 대한 VD 노드 정보

종 류	내 용
ID	View_USEDCAR1
View Deifinition	그림 4의 내용
SEI	< 1, $U, NULL, USEDCAR, //USEDCAR, ISE > < 2, $D, $U, DEALDER, //USEDCAR/DEALER, ISE > < 3, $A, $D, ADDRESS, //USEDCAR/DEALER/ADDRESS, ISE > < 4, $C, $D, CAR, //USEDCAR/DEALER/CAR, ISE > < 5, $P, $C, OPTION, //USEDCAR/DEALER/CAR/OPTION, ISE > < 6, NULL, $D, NAME, //USEDCAR/DEALER/NAME, ESE > < 7, NULL, $D, EMAIL, //USEDCAR/DEALER/EMAIL, ESE > < 8, NULL, $D, PHONE, //USEDCAR/DEALER/PHONE, ESE > < 9, NULL, $C, MAKE, //USEDCAR/DEALER/CAR/MAKE, ESE > < 10, NULL, $C, MODEL, //USEDCAR/DEALER/CAR/MODEL, ESE > < 11, NULL, $C, YEAR, //USEDCAR/DEALER/CAR/YEAR, ESE > < 12, NULL, $P, STATUS, //USEDCAR/DEALER/CAR/OPTION/STATUS, ESE >
CEI	< 1, $A, CITY, //USEDCAR/DEALER/ADDRESS/CITY, ER, =, LA > < 2, $C, PRICE, //USEDCAR/DEALER/CAR/PRICE, ER, <=, 30000 > < 3, $C, DRIVE_MILE, //USEDCAR/DEALER/CAR/DRIVE_MILE, ER, <=, 50000 > < 4, $P, POWERWINDOW, //USEDCAR/DEALER/CAR/OPTION/POWERWINDOW, E, NULL, NULL > < 5, $P, STEREO, //USEDCAR/DEALER/CAR/OPTION/STEREO, ER, =, Radio/Cassette/CD > < 6, $P, AUTOMATIC, //USEDCAR/DEALER/CAR/OPTION/AUTOMATIC, NE, NULL, NULL > < 7, $P, Interiors, //USEDCAR/DEALER/CAR/OPTION/Interiors, NER, =, Leather >
CES	< 1, CEI[1], AND, CEI[2] > < 2, CES[1], AND, CEI[3] > < 3, CES[2], AND, CEI[4] > < 4, CES[3], AND, CEI[5] > < 5, CES[4], AND, CEI[6] > < 6, CES[5], AND, CEI[7] >
VRS	< 1, /View_USEDCAR1, NULL > < 2, /View_USEDCAR1/CAR, NULL > < 3, /View_USEDCAR1/CAR/NAME, {6} > < 4, /View_USEDCAR1/CAR/EMAIL, {7} > < 5, /View_USEDCAR1/CAR/PHONE, {8} > < 6, /View_USEDCAR1/CAR/MAKE, {9} > < 7, /View_USEDCAR1/CAR/MODEL, {10} > < 8, /View_USEDCAR1/CAR/YEAR, {11} > < 9, /View_USEDCAR1/CAR/STATUS, {12} >

Result Structure)는 뷰 인덱스로부터 뷰 결과를 실체화할 때 사용되는 정보로서 뷰 결과의 구조 및 관련 ESE를 저장한다. 표 4는 그림 2의 XML 데이타에 대한 그림 4의 XQuery 뷰 정의로부터 얻어지는 VD 정

표 5 VIE 노드 구조 및 정보 (구성에서 * 표시는 0 또는 그 이상 횟수의 반복을 의미)

종 류	구 성	설 명
EID	<EID>	엘리먼트의 EID
Type	<Type> - Type : { ESE | ISE | NSE }	VIE 노드 종류
SEIndex	<SEIndex>	VD 노드의 SEI 배열 인덱스
REI (Relevant Element Information)	<<CEIndex, SF> : <EID, Value?>*>* - CEIndex : CEI 배열 인덱스 - SF : 조건 만족 여부 (Satisfication Flag) - EID : RE 엘리먼트 ID - Value : CEI[CEIndex].CT가 ER 또는 NER인 경우 RE 엘리먼트 내용	연관성 검사를 위한 정보
Parent VIE	<ParentVIE>	부모 VIE 노드 포인터
Sibling VIE	<SiblingVIE>	형제 VIE 노드 포인터
First Child VIE	<ChildVIE>	첫 번째 자식 VIE 노드 포인터

보를 나타낸 것이다.

VIE 노드의 종류는 **ESE VIE 노드, ISE VIE 노드, NSE(Non-Selected Element) VIE 노드**로 나눌 수 있다. ESE VIE 노드는 VD 노드의 SEI 중 ESE에 해당하는 엘리먼트로 구성되는 VIE 노드 중 뷰 조건을 만족하는 엘리먼트로 구성된 VIE 노드이다. ISE VIE 노드는 VD 노드의 SEI 중 ISE에 해당하는 엘리먼트로 구성되는 VIE 노드 중 뷰 조건을 만족하는 엘리먼트로 구성된 VIE 노드이다. NSE VIE 노드는 뷰 조건을 만족하지 않는 VIE 노드에 해당된다.

REI(Relevant Element Information)는 VIE 노드에 저장되는 정보로서 해당 VIE 노드 및 그 하위 VIE

```
Make_View_Index(View_Definition_Node VD)
{
    - VIECQ : view index element calculation query
    - SEL : selected element list without WHERE clause ordered by DFS
        - item structure : <ParentEID, SEIndex, EID>*
    - REICQ : relevant element information calculation query
    - REL : relevant element list
        - item structure : <ParentEID, CEIndex, EID, Value>*

    // VD 노드 정보를 참조하여 VIE 노드를 구성할 엘리먼트 리스트를 얻어오기 위한 질의를 생성
    VIECQ ← Make_VIE_Calculation_Query(VD);
    // 위에서 만들어진 질의를 하부 데이타 소스에 보내어 반환된 결과를 참조하여 SEL를 생성
    SEL ← Evaluate_VIE(VIECQ);

    // VD 노드 정보를 참조하여 VIE 노드의 연관성 정보를 얻어오기 위한 질의를 생성
    REICQ ← Make_REI_Calculation_Query(VD);
    // 위에서 만들어진 질의를 하부 데이타 소스에 보내어 반환된 결과를 참조하여 REL를 생성
    REL ← Evaluate_REI(REICQ);

    for s in SEL do
        Add_View_Index_Element(VD, s);      // 각 VIE 노드 생성

    for r in REL do
        Add_Relevant_Element(VD, r);        // 해당 VIE 노드에 연관성 정보 삽입

    for v in VIE do
        for r in v.REI do
        r.SF ← Evaluation_REI(VD, r);       // 각 조건(REI) 평가

    EvaluationView(VD);
        - VIE 노드의 Type 결정 : VD 노드의 CES를 참조하여 전체 조건를 평가하여 VIE 노드의 뷰 만족 여부 결정
}
```

그림 6 XML 뷰 인덱스 생성 알고리즘

노드의 뷰 만족 여부에 영향을 주는 엘리먼트(RE: Relevant Element)에 대한 정보이다. RE 엘리먼트 정보는 해당 VIE 노드의 뷰 만족 여부를 평가하기 위하여 엘리먼트 식별자뿐만 아니라 VD 노드의 CEI 번호와 조건 형태에 따른 엘리먼트 값 등으로 구성된다. RE 엘리먼트 정보가 저장되는 위치는 RE 엘리먼트의 부모 엘리먼트에 해당하는 VIE 노드에 저장된다. VIE 노드에 저장되는 자세한 정보는 표 5와 같다.

4.2 XML 뷰 인덱스 생성 알고리즘

본 논문에서 제안하는 XML 뷰 인덱스를 생성하는 과정은 크게 VD 노드를 구성하는 과정과 각 VIE 노드를 구성하는 과정으로 나눌 수 있다. VD 노드를 구성하는 과정은 뷰 정의의 파싱 과정을 통하여 어렵지 않게 이루어진다. 따라서, 본 논문에서는 VD 노드를 구성하는 자세한 과정은 다루지 않는다. 각 VIE 노드를 구성하는 알고리즘은 그림 6과 같으며 다음과 같은 순서로 수행된다.

1. VIE 노드를 구성할 VD 노드의 SEI에 해당하는 모든 엘리먼트 정보(SEL)를 얻어온다.
2. VIE 노드에 저장될 VD 노드의 CEI에 해당하는 모든 엘리먼트 정보(REL)를 얻어온다.
3. SEL을 이용하여 모든 VIE 노드를 만든다. (VIE 노드 생성 및 EID, SEIndex 저장)
4. REL의 각 엔트리를 해당 VIE 노드의 REI에 저장한다. (VIE 노드의 REI 정보 저장)
5. VIE 노드의 각 REI에 해당하는 조건을 평가한다.(각 REI의 SF 결정)
6. VD 노드의 CES를 참조하여 뷰의 조건을 평가하여 각 VIE 노드의 타입을 결정한다. (VIE 노드의 Type 결정)

표 6은 그림 6의 뷰 인덱스 생성 알고리즘이 그림 2의 XML 데이타와 그림 4의 뷰 정의에 대한 XML 뷰 인덱스의 생성 과정에서 도출한 SEL과 REL을 나타낸 것이며, 그림 7과 표 7은 각각 동일 예에서 생성된 뷰 인덱스와 VIE 노드의 REI를 나타낸 것이다.

표 6 그림 2/그림 4 예에 대하여 뷰 인덱스 생성 알고리즘이 도출한 SEL 및 REL

SEL	REL
<NULL, 1(USEDCAR), 1>	<6, 1(CITY), 8, LA>
<1, 2(DEALER), 2>	<10, 2(PRICE), 14, 29500>
<2, 6(NAME), 3>	<10, 3(DRIVE_MILE), 15, 45000>
<2, 7(EMAIL), 4>	<16, 4(POWERWINDOW), 18, Yes>
<2, 8(PHONE), 5>	<16, 5(STEREO), 20, Radio/Cassette/CD>
<2, 3(ADDRESS), 6>	<16, 6(AUTOMATIC), NULL, NULL>
<2, 4(CAR), 10>	<16, 7(Interiors), 21, Leather>
<10, 9(MAKE), 11>	<28, 1(CITY), 30, LA>
<10, 10(MODEL), 12>	<32, 2(PRICE), 36, 31000>
<10, 11(YEAR), 13>	<32, 3(DRIVE_MILE), 37, 45000>
<10, 5(OPTION), 16>	<38, 4(POWERWINDOW), 41, Yes>
<16, 12(STATUS), 23>	<38, 5(STEREO), 40, Radio/Cassette/CD>
<1, 2(DEALER), 24>	<38, 6(AUTOMATIC), NULL, NULL>
<24, 6(NAME), 25>	<38, 7(Interiors), 42, Leather>
<24, 7(EMAIL), 26>	
<24, 8(PHONE), 27>	
<24, 3(ADDRESS), 28>	
<24, 4(CAR), 32>	
<32, 9(MAKE), 33>	
<32, 10(MODEL), 34>	
<32, 11(YEAR), 35>	
<32, 5(OPTION), 38>	
<38, 12(STATUS), 44>	

4.3 XML 뷰 실체화 알고리즘

뷰 인덱스 구조로부터의 XML 뷰 실체화 과정은 뷰 인덱스에 저장되어 있는 각 ESE VIE 노드의 EID를 데이타 소스로 보내어 EID에 해당하는 엘리먼트 내용을 얻어오는 과정을 반복하여 이를 통합하는 것이다. XML 뷰 실체화 과정을 효율적으로 지원하기 위하여 하부 데이타 소스는 3.1절에서 설명한 바와 같이 EID를 통하여

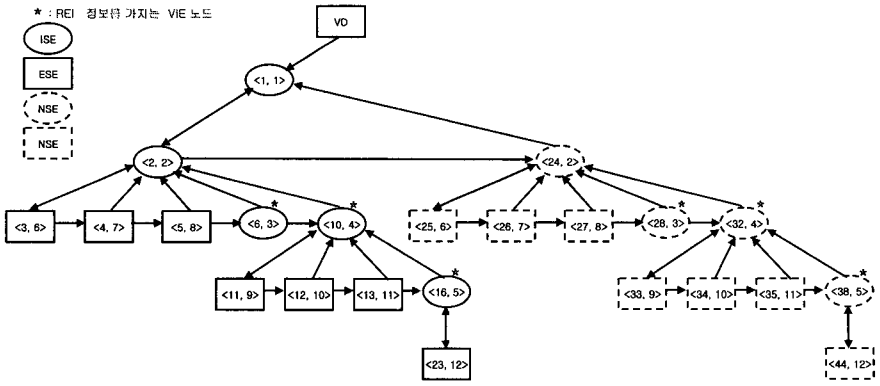

그림 7 그림 2/그림 4 예에 대한 뷰 인덱스 구조 : <EID, SEIndex>

표 7 그림 2/그림 4 예에 대한 뷰인덱스의 각 VIE 노드의 REI 정보

VIE 노드의 EID	REI
6	<1, T> : {<8, LA>}
10	<2, T> : {<14, 29500>} <3, T> : {<15, 45000>}
16	<4, T> : {<18, NULL>} <5, T> : {<20, Radio/Cassette/CD>} <6, T> : {} <7, T> : {<21, Leather>}
28	<1, T> : {<30, LA>}
32	<2, F> : {<36, 31000>} <3, T> : {<37, 45000>}
38	<4, T> : {<41, NULL>} <5, T> : {<40, Radio/Cassette/CD>} <6, T> : {} <7, T> : {<42, Leather>}

해당 엘리먼트의 내용을 바로 얻어올 수 있는 기능을 지원해야 한다. 뷰 인덱스의 ESE VIE 노드에 해당하는 EID를 데이타 소스로 보내는 순서는 DFS (Depth-First-Search)를 따른다. 그림 8은 XML 뷰 실체화 알고리즘을 나타낸 것이다.

```
View_Materialization(View_Definition_Node VD)
{
    EID_List : ESE element ID list ordered by DFS
        - item structure : <SEIndex, EID>*
    EContent_List : element content list
        - item structure : <SEIndex, EID, EContent>*
    // DFS 순서로 ESE VIE 노드를 traverse하면서 구성
    EID_List ← GetAllESE(VD);
    // 하부 데이타 소스로부터 각 EID에 해당하는 엘리먼트 내용을 얻어옴
    EContent_List ← GetElementContent(EID_List);
    // VD 노드의 VRS를 참조하여 뷰 결과 구성
    viewResult ← MakeViewResult(VD, EContent_List);
    return viewResult;
}
```

그림 8 XML 뷰 실체화 알고리즘

5. XML 뷰 인덱스의 점진적 갱신

뷰 인덱스의 점진적 갱신은 먼저 하부 데이타의 변경이 뷰 인덱스에 영향을 주는 지를 파악한 후(연관성 검사), 영향을 준다고 판단되는 경우 하부 데이타의 변경을 뷰 인덱스에 반영하는 것으로 이루어진다.

5.1 연관성 검사

데이타 소스에서의 변경 연산은 엘리먼트 추가(INSERT), 삭제(DELETE), 값 변경(CHANGE) 등으로 나눌 수 있다. 각 변경 연산의 종류 및 그 내용에 따라 뷰 인덱스 구조에 영향을 주는 지를 판단하는 연관성 검사를 통하여 VIE 노드 추가, 삭제, 무변화 등의 조치를 취하게 된다. 연관성 검사는 데이타 소스에서의 변경이 뷰 인덱스에 영향을 주는 지의 여부뿐만 아니라 어떤 VIE 노드에 어떤 영향을 주는 지도 알아내야 한다.

표 8은 각 변경 연산에 따라 수행하는 연관성 검사 항목과 그 결과를 나타낸 것이다. 그림 9는 표 8의 연관성 검사 항목 및 결과에 따라 수행되는 연관성 검사 알고리즘을 나타낸 것이다. 표 8 및 그림 9에서 나타낸 바와 같이 VIE 노드가 추가되는 경우는 SEI에 해당하는 엘리먼트가 추가된 경우만 해당된다. 이 경우에는 하부 XML 데이타 소스에 뷰 유지 문장을 제기하여 뷰 인덱스에 대한 갱신을 수행해야 한다. VIE 노드가 삭제되는 경우는 SEI에 해당하는 엘리먼트가 삭제된 경우만 해당된다. 이 경우에는 해당 VIE 노드 및 하위 VIE 노드를 삭제하면 된다. VIE 노드의 종류가 변경되는 경우는 추가/삭제/값 변경된 엘리먼트가 CEI에 해당하는 엘리먼트이고, 이로 인해 해당 VIE 노드의 해당 REI의 SF 값이 변하는 경우이다. 이 경우에는 추가/삭제/값 변경된 엘리먼트를 해당 VIE 노드의 REI에 추가/삭제/값 변경한 후 SF를 재평가하여 그 값이 변경되는 지의 여부를 살펴본 후, 변경된 경우 이와 관련된 VIE 노드를 재평

표 8 연관성 검사 항목 및 결과

변경 연산 종류 / 뷰 인덱스 영향 종류	추가 연산	삭제 연산	값 변경 연산 (∈ CEP이고, CT가 NER 또는 ER인 경우)
VIE 노드 추가	* ∈ SEP	-	-
VIE 노드 삭제	-	* ∈ SEP	-
VIE 노드 종류변경	* ∈ CEP - 만족 → 불만족 - 불만족 → 만족	* ∈ CEP - 불만족 → 만족 - 만족 → 불만족	* Case 1 - OldValue : 불만족 - NewValue : 만족 * Case 2 - OldValue : 만족 - NewValue : 불만족
무변화	* ∉ SEP * ∉ CEP * ∈ CEP - 불만족 → 불만족 - 만족 → 만족	* ∉ SEP * ∉ CEP * ∈ CEP - 불만족 → 불만족 - 만족 → 만족	* Case 1 - Old/NewValue : 불만족 * Case 2 - Old/NewValue : 만족

```
Relevance_check(View_Definition_Node V, Update_Operation U)
{
    View_Index_Element_Node vie ← null;

    if (IsContain(CEP(VD), EPath(U)) { // 조건절에 나타나는 엘리먼트인 경우
        CEIndex ← CEP가 EPath(U)에 해당하는 CEI 배열 인덱스
        CT ← CEI[CEIndex].CT;
        vie ← <EID(U), *>를 REI에 저장하고 있는 VIE 노드;
        if (Type(U) == INSERT) {
            Add_Relevant_Element(vie, CEIndex, U);
                - VIE 노드의 CEIndex에 해당하는 REI에 <EID(U), Value(U)> 또는 <EID(U), NULL> 추가
        }
        else if (Type(U) == DELETE) {
            Remove_Relevant_Element(vie, CEIndex, U);
                - VIE 노드의 CEIndex에 해당하는 REI로부터 <EID(U), *> 제거
        }
        else if (Type(U) == CHANGE) {
            if ((CT == ER) || (CT == NER)) {
                Replace_Relevant_Element(vie, CEIndex, U);
                    - VIE 노드의 CEIndex에 해당하는 REI의 <EID(U), *>를 <EID(U), Value(U)>로 교체
            }
            else {
                vie ← null;
            }
        }
        if (vie != null) {
            rei ← VIE 노드의 CEIndex에 해당하는 REI;
            old_sf ← rei.SF;
            new_sf ← Evaluation_REI(rei);
            if (old_sf == new.sf) {
                vie ← null;
            }
        }
    }

    Incremental_Update(VD, U, vie);
}
```

그림 9 연관성 검사 알고리즘

```
Incremental_Update(View_Definition_Node VD, Update_Operation U, View_Index_Element_Node vie)
{
    if (IsContain(SEP(VD), EPath(U)) { // U의 EPath가 SEP의 조상 또는 자신인 경우
        if (Type(U) == INSERT) {
            // 뷰 인덱스 생성 알고리즘에서와 같이 데이타 소스에 서브 질의를 보내어 (SEL, REL) 정보 얻어오기.
            // (SEL, REL)를 통하여 뷰 인덱스 재구성
        }
        else if (Type(U) == DELETE) {
            // EID(U)에 해당하는 VIE 노드 및 그 하위 VIE 노드를 삭제
        }
    }
    else {
        if (vie == null) return;
        Evaluation_SubView(VD, vie);
            - vie와 연관된 서브 뷰 인덱스의 VIE 노드의 뷰 만족 여부 재계산 (VIE 노드의 Type 결정)
    }
}
```

그림 10 점진적 갱신 알고리즘

가하여 그 종류를 다시 할당하면 된다. 추가/삭제/값 변경된 엘리먼트가 SEI 및 CEI에 해당되는 엘리먼트가 아니거나, CEI에 해당되는 엘리먼트라 하더라도 해당 VIE 노드의 해당 REI의 SF 값이 변하지 않는 경우는 뷰 인덱스에 VIE 노드 추가/삭제/노드 종류 변경의 영향을 미치지 않는다. 이 경우에는 변경 연산을 무시하거나 CEP에 해당하는 엘리먼트인 경우에는 해당 VIE 노드의 REI에 추가/삭제/값 변경만 해주면 된다.

5.2 갱신 알고리즘

본 논문에서의 갱신 알고리즘은 그림 10과 같이 수행된다. 데이타 소스에서의 변경 연산이 선택절에 나타나는 엘리먼트의 추가인 경우에는 데이타 소스로 VIE 노드 추가를 위한 질의를 생성하여 보내게 된다. VIE 노드 추가를 위한 질의는 뷰 인덱스 생성 알고리즘에서와 같이 데이타 소스로부터 VIE 노드에 해당하는 엘리먼트 정보(SEL)뿐만 아니라 그 VIE 노드에 영향을 주는 엘리먼트 정보(REL)를 얻어오기 위한 것이다. 데이타 소스로부터 정보가 반환되면 뷰 인덱스 생성 알고리즘에서와 같이 뷰 인덱스에 VIE 노드를 생성하고 REI를 저장한 후, 각 VIE 노드의 종류를 결정한다. 데이타 소스에서의 변경 연산이 선택절에 나타나는 엘리먼트의 삭제인 경우에는 데이타 소스에서 삭제된 엘리먼트에 해당하는 VIE 노드 및 그 하위 VIE 노드를 삭제하기만 하면 된다. 그 외의 경우에는 그림 9의 연관성 검사 알고리즘에 의해 데이타 소스에서의 변경 연산이 뷰 인덱스에 영향을 준다고 결정된 경우에는 해당 VIE 노드와 연관된 VIE 노드들에 대하여 뷰 만족 여부를 판단하여 VIE 노드들의 종류를 다시 할당하면 된다.

5.3 점진적 갱신 예

본 절에서는 5.1 절과 5.2 절에서 설명한 연관성 검사와 점진적 갱신 알고리즘을 여러 가지 형태의 변경 연산에 적용하였을 경우 그림 2/그림 4 예에서의 뷰 인덱스의 갱신 예를 살펴본다. 첫번째 예는 조건절에 나타나는 엘리먼트의 값을 변경한 경우이고, 두번째 예는 선택절에 나타나는 엘리먼트가 추가된 경우이며, 세번째 예는 조건절에 나타나는 엘리먼트가 추가된 경우이다.

5.3.1 조건절에 나타나는 엘리먼트의 값 변경

변경 연산 U_1(<CHANGE, 32, PRICE, 36, //USEDCAR/DEALER/CAR/PRICE, 30000>)에 대해서 그림 9의 연관성 검사 알고리즘을 적용하면, EPath(U_1)가 VD.CEI[2]의 CEP에 해당하고, Type(U_1)이 CHANGE이고, VD.CEI[2].CT가 ER이므로, VIE 노드 32의 REI[1]의 <36, 31000>을 <36, 30000>으로 수정한 후, REI[1]을 재평가하면 REI[1].SF가 False에서 True로 변경된다. 따라서, VIE 노드 32에 점진적 갱신 알고리즘을 적용한다. 점진적 갱신 알고리즘에서 VIE 노드 32와 관련된 VIE 노드 28, 32, 38에 저장된 REI를 가지고 VD 노드의 CES를 참조하여 조건을 평가하면 True이므로, VIE 노드 24, 25, 26, 27, 28, 32, 33, 34, 35, 38, 44의 VIE 노드 종류를 VD 노드의 SEI의 Type으로 바꾼다. 그림 11과 표 9는 각각 변경 연산 U_1을 적용한 후의 뷰 인덱스와 각 VIE 노드의 REI 정보를 나타낸다.

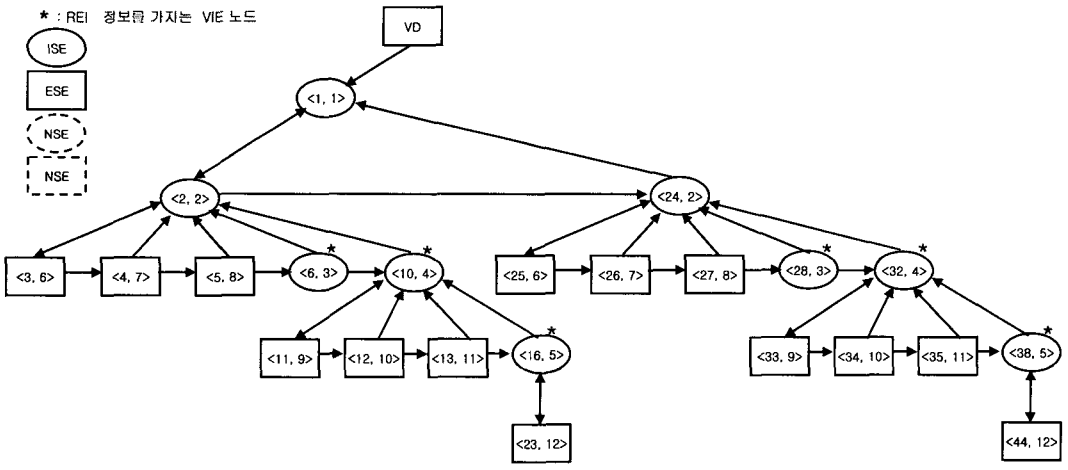

그림 11 변경 연산 U_1을 적용한 후의 뷰 인덱스

표 9 변경 연산 U_1을 적용한 후의 각 VIE 노드의 REI 정보

VIE 노드의 EID	REI
6	<1, T> : {<8, LA>}
10	<2, T> : {<14, 29500>} <3, T> : {<15, 45000>}
16	<4, T> : {<18, NULL>} <5, T> : {<20, Radio/Cassette/CD>} <6, T> : {} <7, T> : {<21, Leather>}
28	<1, T> : {<30, LA>}
32	**<2, T> : {<36, 30000>}** <3, T> : {<37, 45000>}
38	<4, T> : {<41, NULL>} <5, T> : {<40, Radio/Cassette/CD>} <6, T> : {} <7, T> : {<42, Leather>}

```
<CAR EID="45">
  <MAKE EID="46">Mercedes Benz</MAKE>
  <MODEL EID="47">M-Class</MODEL>
  <YEAR EID="48">1998</YEAR>
  <PRICE EID="49">29500</PRICE>
  <DRIVE_MILE EID="50">45000</DRIVE_MILE>
  <OPTION EID="51">
    <POWERLOCK EID="52">Yes</POWERLOCK>
    <POWERWINDOW EID="53">Yes</POWERWINDOW>
    <Alarm EID="54">Yes</Alarm>
    <STEREO EID="55">Radio/Cassette/CD</STEREO>
      <Interiors EID="56">Leather</Interiors>
    <AIRCONDITIONER EID="57">Yes</AIRCONDITIONER>
    <STATUS EID="58">Very nice pre-owned</STATUS>
  </OPTION>
</CAR>
```

그림 12 변경 연산 U_2에 대한 하부 데이타

표 10 변경 연산 U_2에 대한 SEL 및 REL

SEL	REL
<2, 4(CAR), 45> <45, 9(MAKE), 46> <45, 10(MODEL), 47> <45, 11(YEAR), 48> <45, 5(OPTION), 51> <51, 12(STATUS), 58>	<45, 2(PRICE), 49, 29500> <45, 3(DRIVE_MILE), 50, 45000> <51, 4(POWERWINDOW), 53, Yes> <51, 5(STEREO), 55, Radio/ Cassette/CD> <51, 6(AUTOMATIC), NULL, NULL> <51, 7(Interiors), 56, Leather>

5.3.2 선택절에 나타나는 엘리먼트의 추가

그림 12와 같은 CAR 엘리먼트 추가에 대한 변경 연산 U_2(<INSERT, 2, CAR, 45, //USEDCAR/DEALER/CAR, NULL>)에 대해서는 Type(U_2)는 INSERT이고, EPath(U_2)가 VD.SEI[4].SEP에 해당하므로 데이타 소스로부터 표 10과 같은 SEL, REL 정보를 얻어와서 뷰 인덱스 생성 알고리즘과 같이 뷰 인덱스에 추가한 후, 추가된 각 VIE 노드의 REI를 평가하고, 추가된 VIE 노드와 관련된 뷰의 조건에 해당하는 VIE 노드 6, 45, 51에 저장된 REI를 가지고 VD 노드의 CES를 참조하여 조건을 평가하면 True이므로, VIE 노드 45, 46, 47, 48, 51, 58의 VIE 노드 종류를 VD 노드의 SEI의 Type으로 할당한다. 그림 13과 표 11은 각각 변경 연산 U_2를 적용한 후의 뷰 인덱스와 각 VIE 노드의 REI 정보를 나타낸다.

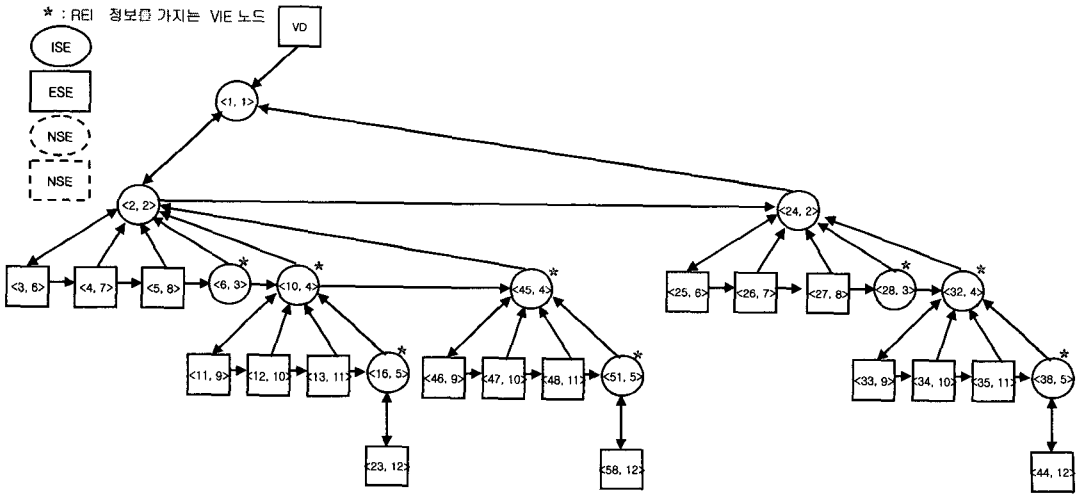

그림 13 변경 연산 U_2를 적용한 후의 뷰 인덱스

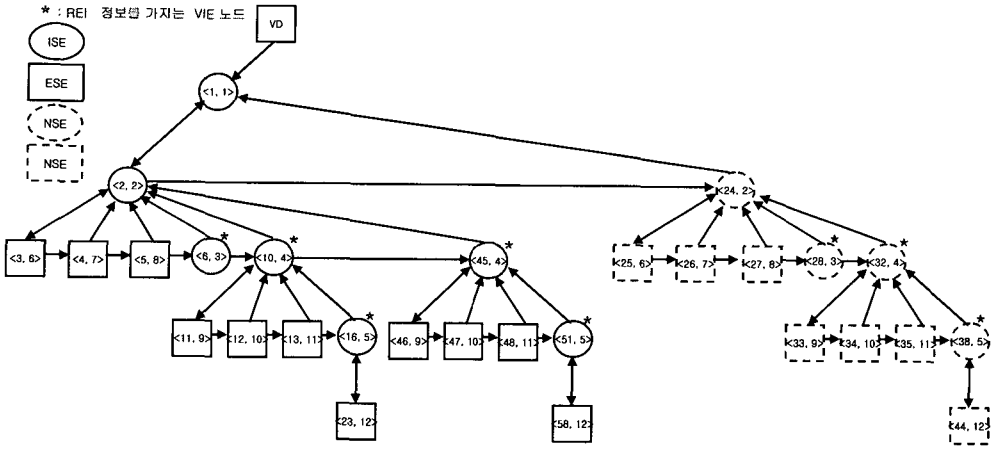

그림 14 변경 연산 U_3를 적용한 후의 뷰 인덱스

표 11 변경 연산 U_2를 적용한 후의 각 VIE 노드의 REI 정보

VIE 노드의 EID	REI
6	<1, T> : {<8, LA>}
10	<2, T> : {<14, 29500>} <3, T> : {<15, 45000>}
16	<4, T> : {<18, NULL>} <5, T> : {<20, Radio/Cassette/CD>} <6, T> : {} <7, T> : {<21, Leather>}
45	**<2, T> : {<49, 29500>}** **<3, T> : {<50, 45000>}**
51	**<4, T> : {<53, NULL>}** **<5, T> : {<55, Radio/Cassette/CD>}** **<6, T> : {}** **<7, T> : {<56, Leather>}**
28	<1, T> : {<30, LA>}
32	<2, T> : {<36, 30000>} <3, T> : {<37, 45000>}
38	<4, T> : {<41, NULL>} <5, T> : {<40, Radio/Cassette/CD>} <6, T> : {} <7, T> : {<42, Leather>}

표 12 변경 연산 U_3를 적용한 후의 각 VIE 노드의 REI 정보

VIE 노드의 EID	REI
6	<1, T> : {<8, LA>}
10	<2, T> : {<14, 29500>} <3, T> : {<15, 45000>}
16	<4, T> : {<18, NULL>} <5, T> : {<20, Radio/Cassette/CD>} <6, T> : {} <7, T> : {<21, Leather>}
45	<2, T> : {<49, 29500>} <3, T> : {<50, 45000>}
51	<4, T> : {<53, NULL>} <5, T> : {<55, Radio/Cassette/CD>} <6, T> : {} <7, T> : {<56, Leather>}
28	<1, T> : {<30, LA>}
32	<2, T> : {<36, 30000>} <3, T> : {<37, 45000>}
38	<4, T> : {<41, NULL>} <5, T> : {<40, Radio/Cassette/CD>} **<6, F> : {<59, NULL>}** <7, T> : {<42, Leather>}

5.3.3 조건절에 나타나는 엘리먼트의 추가

본 절의 예는 조건절에 나타나는 엘리먼트의 추가에 대한 변경 연산 U_3(<INSERT, 38, AUTOMATIC, 59, //USEDCAR/DEALER/CAR/OPTION/AUTOMATIC, Yes>)에 대하여 연관성 검사 및 점진적 갱신을 수행하는 경우이다. EPath(U_3)가 VD.CEI[6]의 CEP에 해당하고, Type(U_3)이 INSERT이고, VD.CEI[2].CT가 NE이므로, VIE 노드 38의 REI[3]에 <59, NULL>을 추가한 후, REI[3]을 재평가하면 REI[3].SF가 T에서 F로 변경된다. 따라서, VIE 노드 38에 점진적 갱신 알고리즘을 적용한다. 점진적 갱신 알고리즘에서 VIE 노드 38와 관련된 VIE 노드 28, 32, 38에 저장된 REI를 가지고 VD 노드의 CES를 참조하여 조건을 평가하면 False이므로, VIE 노드 24, 25, 26, 27, 28, 32, 33, 34, 35, 38, 44의

VIE 노드 종류를 NSE로 바꾼다. 그림 14와 표 12는 각각 변경 연산 U_3를 적용한 후의 뷰 인덱스와 각 VIE 노드의 REI 정보를 나타낸다.

6. 구현 및 성능 평가

본 논문에서는 성능 평가를 위하여 XML 뷰 인덱스 구조와 이와 관련된 알고리즘을 구현하여 XML 뷰 관리 시스템을 구성하였다. 본 장에서는 구현 및 성능 평가 결과를 기술한다. 구현된 XML 뷰 관리 시스템의 동작을 위해 필요한 XML 저장 관리 시스템의 기능 중 XQuery 질의 처리 기능은 XQuery 특징을 거의 모두 수용하여 구현된 Kweelt[31]라는 XQuery 엔진을 사용하였으며, 엘리먼트 식별자를 이용한 엘리먼트 내용 추출 기능 및 하부 XML 데이타의 변경 통보 기능은 시뮬레이션하였다.

본 성능 평가는 본 논문에서 제안한 XML 뷰 인덱싱(VI)[1] 기법과 XML 실체뷰(MV)[2] 기법을 비교 평가하는데 주력하였으며, 필요한 경우 XML 뷰를 유지하지 않는 경우(NoVM: No View Maintenance)[3] 즉, 질의 결과를 뷰 인덱스나 실체뷰로 유지하지 않는 경우와도 비교하였다. MV의 경우, 그 구조는 뷰 갱신을 위한 부분에서는 XML 뷰 인덱스 구조와 동일한 구조를 가지지만 추가로 뷰를 실체화한 엘리먼트 내용을 가진다. 이 차이 때문에 MV 관련 알고리즘들은 본 논문에서 제안한 VI 관련 알고리즘을 약간 수정하여 구현하였다.

6.1 성능 평가 척도

본 성능 평가에서 본 논문에서 제안한 XML 뷰 인덱싱 기법과 비교 평가를 위한 XML 실체뷰 기법의 뷰 구조는 4.1절의 XML 뷰 인덱스 구조와 동일하며, 엘리먼트 내용을 실체화하기 위하여 VIE 노드 구조에 엘리먼트 내용을 저장하기 위한 Value 필드를 추가한 구조이다. 본 논문에서 제안한 XML 뷰 인덱싱 기법에서는 하부 XML 데이타 소스에 대한 접근을 최소화하기 위해 뷰를 만족하지 않는 엘리먼트에 대한 뷰 관리 정보(연관성 검사 및 갱신 정보)도 유지한다. 따라서, MV에서도 뷰를 만족하지 않는 엘리먼트에 대한 뷰 관리 정보 및 엘리먼트 내용을 유지한다. MV에서 뷰 구조 생성 시 엘리먼트 내용을 얻어오는 방법은 그림 6에서 SEL을 얻어올 때 VI에서의 VIECQ(SEL을 얻어오기 위한 XQuery 질의)를 엘리먼트 내용도 함께 얻어올 수 있도록 수정한 질의로 대체함으로써 이루어진다.

본 성능 평가는 1) 초기 뷰 질의 처리 시간, 2) XML 뷰 구조 생성 시간, 3) XML 뷰 결과 생성 시간, 4) XML 뷰 갱신 시간, 5) XML 뷰 저장 공간 등으로 나누어 수행하였다. 첫째, 초기 뷰 질의 처리 시간은 XML 뷰 구조 생성 시간과 XML 뷰 결과 생성 시간으로 이루어진다. 초기 뷰 질의 처리 시간의 평가는, 뷰 질의가 처음 제기되었을 때 이를 처리하는 시간을 VI, MV, NoVM 간에 비교하기 위한 것이다. 뷰 검색 결과를 향후 관리하지 않는 NoVM에 비해 VI나 MV는 XML 뷰 구조를 생성 및 저장해야 하므로 이에 따른 오버헤드가 발생한다. 둘째, XML 뷰 구조 생성 시간은 VI의 경우에는 그림 6의 XML 뷰 인덱스 생성 알고리즘을 수행하는 시간을 말하며, MV의 경우에는 엘리먼트 내용의 실체화를 위해 변경된 알고리즘을 수행하는 시간을 말한다. 셋째, XML 뷰 결과 생성 시간은 VI의 경우에는 뷰 인덱스로부터 뷰 결과를 구성하는 엘리먼트 식별자를 구하여 이를 하부 XML 데이타 소스에 보내어 엘리먼트 내용을 얻어온 후 이를 바탕으로 뷰 결과를 구성하는 데 걸리는 시간(그림 8의 XML 뷰 실체화 알고리즘 수행 시간)을 말하며, MV의 경우는 이미 엘리먼트 내용이 실체화되어 있으므로 따로 실체화 과정이 필요없이 뷰 구조에 저장되어 있는 엘리먼트 내용을 바탕으로 뷰 결과를 구성하는 데 걸리는 시간을 말한다. 넷째, XML 뷰 갱신 시간은 변경 연산에 대한 연관성 검사를 수행하여 이를 바탕으로 XML 뷰 구조를 갱신하는 데 걸리는 시간(그림 9의 연관성 검사 알고리즘 및 그림 10의 점진적 갱신 알고리즘을 수행하는 데 걸리는 시간)을 말한다. 다섯째, XML 뷰 저장 공간은 VI의 경우 XML 뷰 인덱스 구조가 차지하는 공간을 말하며, MV의 경우에는 VI의 저장 공간에 엘리먼트 내용이 차지하는 공간을 합한 것을 말한다.

6.2 시스템 환경 및 파라미터 설정

본 성능 평가는 Windows 2000 Professional 운영체제를 사용하는 펜티엄-4 2.0GHz의 CPU, 512MB의 메인 메모리를 가지는 시스템 상에서 수행되었다. 또한, Java를 이용하여 구현되었다. 실험에 사용된 뷰 질의는 그림 4의 것을 사용하였으며, XML 데이타는 그림 2의 XML 데이타를 기반으로 확장하여 사용하였다. DEALER 엘리먼트 개수는 10개로 고정하였고, CAR 엘리먼트 개수는 500개부터 5000개까지 500개씩 증가시킨 값들로 하였다. 또한, 전체 CAR 엘리먼트 개수 중 뷰 질의를

1) VI : 그래프에서 본 논문에서 제안한 XML 뷰 인덱싱을 사용한 경우를 나타낸다.
2) MV : 그래프에서 XML 실체뷰를 사용한 경우를 나타낸다.
3) NoVM : 그래프에서 XML 뷰를 유지하지 않는 경우를 나타낸다. 이 경우는 모든 그래프에서 초기 뷰 질의 처리 시간을 사용하였다.

만족하는 CAR 엘리먼트의 비율을 나타내는 XML 질의 만족률은 0~100% 사이의 값을 5% 단위로 증가시키면서 실험을 수행하였다. 또한, XML 뷰 인덱스 실체화 시간에 영향을 주는 엘리먼트 식별자를 이용한 엘리먼트 내용 접근 시간은 엘리먼트 당 0.2 ms로 할당하였다.

6.3절에 나타나는 그래프는 CAR 엘리먼트 개수가 5000개인 경우의 XML 질의 만족률에 따른 성능 측정 데이타와 XML 질의 만족률이 5%인 경우의 CAR 엘리먼트 개수에 따른 성능 측정 데이타이다. 각 그래프는 50번 반복 수행한 결과의 평균 값을 나타낸 것이다.

6.3 실험 결과

본 성능 평가에서는 XML 질의 만족률과 CAR 엘리먼트 개수를 변화시키면서 성능 실험을 수행하였다. 그리고, 실험의 종류는 1) 초기 뷰 질의 처리 시간, 2) XML 뷰 구조 생성 시간, 3) XML 뷰 결과 생성 시간, 4) XML 뷰 갱신 시간, 5) XML 뷰 저장 공간 등으로 나누어 수행하였다.

6.3.1 실험 결과 : 초기 뷰 질의 처리 시간

본 절에서는 초기 뷰 질의 처리 시간을 평가한다. 초기 뷰 질의 처리 시간은 처음으로 뷰를 정의하는 XML 질의를 제기하여 그 결과를 리턴받을 때까지의 응답 시간(response time)을 말한다. NoVM의 경우는 하부 데이타 소스에 직접 XML 뷰 질의를 보내어 결과를 얻어오는 데 걸리는 시간이다. 이에 반해 VI 및 MV의 경우는, 질의의 결과를 향후 유지 및 관리하기 위한 XML 뷰 구조 생성 시간과 뷰 결과 생성 시간을 합한 시간이다.

그림 15와 그림 16은 각각 XML 질의 만족률과 CAR 엘리먼트 개수에 따른 초기 뷰 질의 처리 시간을 나타낸 것이다. 그림 15와 그림 16에서 볼 수 있는 것과 같이 전체적으로 VI 및 MV가 NoVM보다 긴 수행 시간이 필요하다. 이는 VI 및 MV에서는 XML 뷰를 생성하기 위한 복잡한 연산을 수행하기 때문이다. VI와 MV를 비교하면 그림 15에서와 같이 동일한 CAR 엘리먼트 개수에 대하여 XML 질의 만족률이 증가함에 따라 VI의 경우가 MV의 경우보다 긴 수행 시간이 걸리는 것을 알 수 있다. 이는 두 경우의 XML 뷰 구조 생성 시간은 별로 차이가 없지만(MV의 경우 VI와 달리 엘리먼트 내용을 가져오는 것이 필요하지만 이는 뷰 구조 생성 알고리즘 수행 중에 하부 XML 저장 관리 시스템에 제기되는 SEL을 구하기 위한 서로 다른 XQuery 질의(VI의 경우 VIECQ, MV의 경우 엘리먼트 내용을 함께 얻어오기 위해 VIECQ를 수정한 질의)를 수행하는 시간이

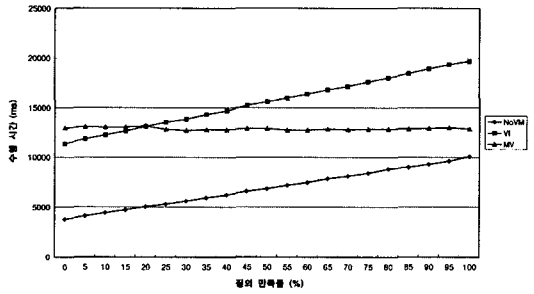

그림 15 XML 질의 만족률에 따른 초기 뷰 질의 처리 시간 (CAR 엘리먼트 개수 = 5000)

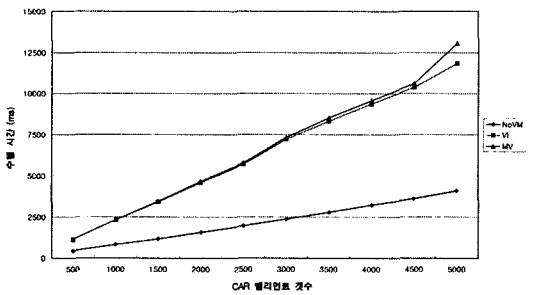

그림 16 CAR 엘리먼트 개수에 따른 초기 뷰 질의 처리 시간 (XML 질의 만족률 = 5%)

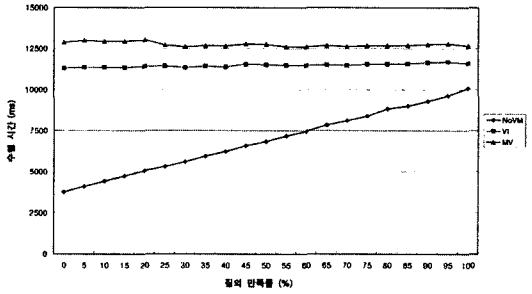

그림 17 XML 질의 만족률에 따른 XML 뷰 구조 생성 시간 (CAR 엘리먼트 개수 = 5000)

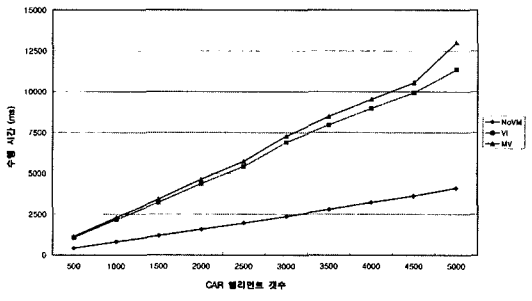

그림 18 CAR 엘리먼트 개수에 따른 XML 뷰 구조 생성 시간 (XML 질의 만족률 = 5%)

므로 실제로 별로 차이가 발생하지 않는다.), XML 뷰 결과 생성 시간은 MV의 경우는 거의 동일한 반면 VI의 경우는 XML 질의 만족률이 증가함에 따라 하부 XML 데이타 소스로부터 개별적으로 엘리먼트 내용을 얻어와야 하는 엘리먼트의 개수가 늘어나기 때문에 XML 질의 만족률이 증가함에 따라 늘어나기 때문이다. 그리고, 그림 16에서와 같이 XML 질의 만족률이 낮은 경우에는 실체화해야 하는 엘리먼트 개수가 적어 XML 뷰 실체화 시간이 적기 때문에 두 경우가 거의 비슷한 정도로 나타난다.

6.3.2 실험 결과 : XML 뷰 구조 생성 시간

본 절에서는 XML 뷰 구조 생성 시간을 평가한다. 그림 17과 그림 18은 각각 XML 질의 만족률과 CAR 엘리먼트 개수에 따른 XML 뷰 구조 생성 시간을 나타낸 것이다. 그림 17에서와 같이 VI와 MV, 두 경우 모두 XML 질의 만족률과는 상관없이 XML 뷰 구조를 생성하는 시간은 거의 동일하다(앞서 설명한 바와 같이 VI와 MV의 XML 뷰 구조 생성 알고리즘의 차이는 SEL을 구하기 위해 서로 다른 VIECQ를 사용하는데 있다.).

하지만, MV의 경우가 엘리먼트 내용을 가져와서 뷰 구조를 구성해야 하기 때문에 VI보다 긴 수행 시간이 필요하다. 그리고, 그림 18에서와 같이 두 경우 모두 XML 질의 대상 엘리먼트 개수가 많아짐에 따라 수행 시간이 증가함을 알 수 있으며, 다소 MV의 경우가 VI의 경우보다 긴 수행 시간이 필요하다.

6.3.3 실험 결과 : XML 뷰 결과 생성 시간

본 절에서는 XML 뷰 결과 생성 시간을 평가한다. 그림 19와 그림 20은 각각 XML 질의 만족률과 CAR 엘리먼트 개수에 따른 XML 뷰 결과 생성 시간을 나타낸 것이다. 그림 19와 그림 20에서와 같이 MV의 경우에는 따로 실체화 과정이 필요없기 때문에 XML 뷰 결과 생성 시간이 거의 일정하며 0에 가까운 것을 알 수 있다. 그러나, 그림 19에서와 같이 VI의 경우에는 XML 질의 만족률이 높아짐에 따라 하부 XML 데이타 소스로부터 엘리먼트 내용을 얻어와야 하는 엘리먼트의 개수가 늘어나기 때문에 XML 뷰 실체화 시간이 증가하여 XML 뷰 결과 생성 시간도 비례적으로 증가함을 알 수 있다. 하지만, 그림 20과 같이 XML 질의 만족률이 낮은 경우

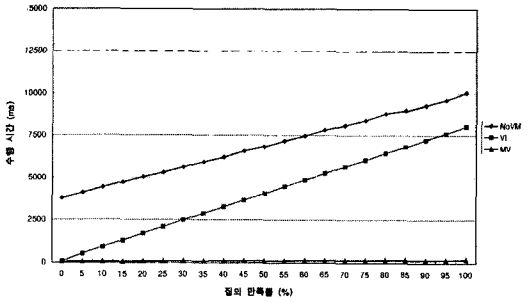

그림 19 XML 질의 만족률에 따른 XML 뷰 결과 생성 시간 (CAR 엘리먼트 개수 = 5000)

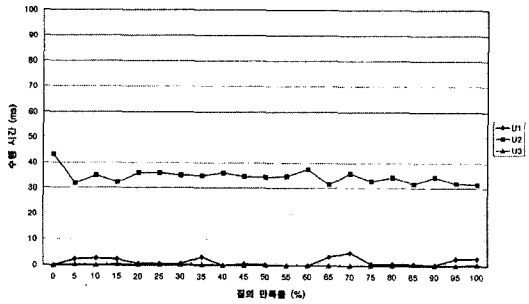

그림 21 XML 질의 만족률에 따른 XML 뷰 인덱스 갱신 시간 (CAR 엘리먼트 개수 = 5000)

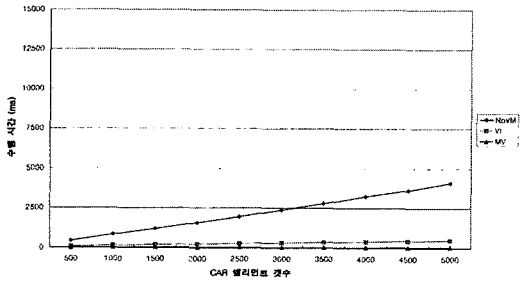

그림 20 CAR 엘리먼트 개수에 따른 XML 뷰 결과 생성 시간 (XML 질의 만족률 = 5%)

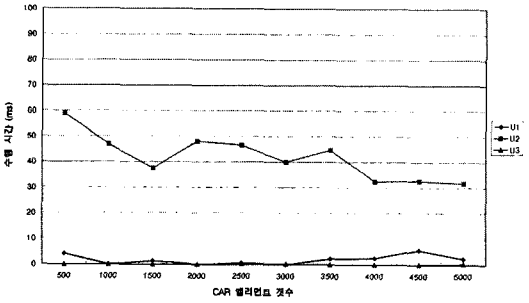

그림 22 CAR 엘리먼트 개수에 따른 XML 뷰 인덱스 갱신 시간 (XML 질의 만족률 = 5%)

에는 실체화해야 하는 엘리먼트 개수가 많지 않기 때문에 MV의 경우와의 차이가 적음을 알 수 있다. 또한, 그림 19와 그림 20에서 나타나듯이 NoVM의 경우보다는 XML 뷰(VI 또는 MV)를 유지하는 것이 반복되는 동일한 질의에 대하여 질의 응답 시간을 줄일 수 있음을 알 수 있다.

6.3.4 실험 결과 : XML 뷰 갱신 시간

본 절에서는 XML 뷰 갱신 시간을 평가한다. 그림 21과 그림 22는 각각 XML 질의 만족률과 CAR 엘리먼트 개수에 따른 5.3절의 점진적 갱신 예를 수행하는 데 걸리는 시간을 나타낸 것이다. 그림 21과 그림 22는 VI의 경우만을 나타낸 것이다. MV의 경우에도 거의 동일한 결과를 나타내며, 실체화를 위한 작업이 필요한 U_2의 경우에만 약간 더 걸린다. 그림 21과 그림 22에서와 같이 XML 질의 만족률과 CAR 엘리먼트 개수에 상관없이 U_1, U_3와 같은 하부 XML 데이타 소스에 XML 뷰 유지 문장을 제기할 필요가 없는 변경 연산의 경우에는 10 ms 이하의 XML 뷰 갱신 시간이 필요하였으며, U_2와 같이 하부 XML 데이타 소스에 XML 뷰 유지 문장을 제기할 필요가 있는 경우에도 약 30 ms~60 ms 정도의 짧은 XML 뷰 갱신 시간이 필요하였다.

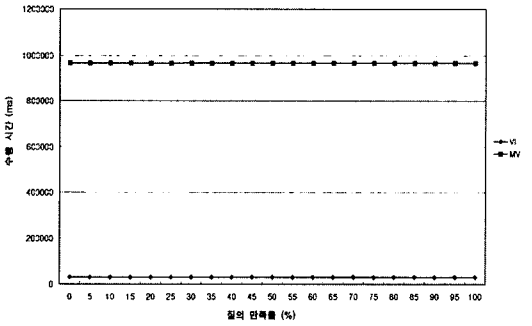

그림 23 XML 질의 만족률에 따른 XML 뷰 저장 공간 크기 (CAR 엘리먼트 개수 = 5000)

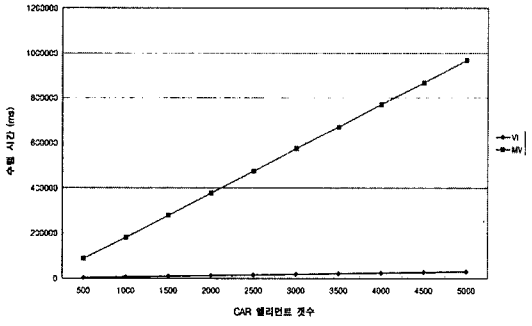

그림 24 CAR 엘리먼트 개수에 따른 XML 뷰 저장 공간 크기 (XML 질의 만족률 = 5%)

6.3.5 실험 결과 : XML 뷰 저장 공간

본 절에서는 XML 뷰 구조 저장 공간 크기를 평가한다. VI와 MV 간의 XML 뷰 구조를 위한 저장 공간의 차이는, 두 기법 모두 뷰 갱신을 위한 정보를 동일하게 가지고 있으므로 MV가 엘리먼트의 실제 내용을 XML 뷰 구조에 포함시킴으로써 생기는 차이이다.

그림 23과 그림 24는 각각 XML 질의 만족률과 CAR 엘리먼트 개수에 따른 VI와 MV의 저장 공간의 크기를 나타낸 것이다. 그림 23에 나타난 것과 같이 VI와 MV는 XML 질의 만족률과 상관없이 일정하다. 왜냐하면, 향후 XML 뷰 유지 시에 하부 XML 데이타 소스에 대한 접근을 최소화하기 위해 뷰 질의를 만족하지 않는 엘리먼트(노드) 정보도 XML 뷰 구조에 포함하여 유지하기 때문이다. 그림 23에서의 MV의 저장 공간 크기는 VI의 저장 공간 크기의 약 32배에 해당된다. 그리고, 그림 24에서와 같이 두 기법 모두 CAR 엘리먼트 개수가 증가함에 따라 비례적으로 증가하지만, 엘리먼트 내용을 포함하는 MV의 기울기가 더 크다. 그림 24에서의 MV의 저장 공간 크기는 VI의 저장 공간 크기의 약 28~32배에 해당된다.

7. 결 론

본 논문에서는 XML 뷰의 구현을 위한 뷰 인덱싱 기법을 제안하였다. XQuery를 이용하여 정의된 XML 질의에 대한 결과를 뷰 형태로 관리하고자 하는 경우 이를 실체화하지 않고 그 결과에 대한 포인터(XML 엘리먼트의 식별자)와 연관성 검사를 위한 정보만을 유지하는 뷰 인덱스를 지원하기 위한 XML 뷰 인덱스 구조 및 뷰 인덱스 생성 알고리즘을 제안하였다. 또한, 하부 데이타 소스에 대한 엘리먼트 단위의 변경을 효율적으로 뷰 인덱스에 반영하기 위한 연관성 검사 알고리즘 및 점진적 갱신 알고리즘을 제안하였다. 아울러 예를 통하여 XML 뷰 인덱싱의 뷰 인덱스 구조와 관련 알고리즘의 동작을 보였다.

본 논문에서 제안한 XML 뷰 인덱싱 기법을 통하여 자주 접근되는 XML 뷰를 실체화하여 유지할 때 발생하는 저장 공간 오버헤드를 줄이면서 XML 질의 결과를 빠르게 구성할 수 있다. 제안된 기법은 XQuery 엔진인 Kweelt를 사용하여 Java 언어로 구현되었다. 성능 평가 결과 XML 뷰 인덱싱 기법이 실체화 시간으로 인해 XML 실체뷰 기법보다 질의 재수행 시간(실험 결과

중 XML 뷰 결과 생성 시간)은 약 2~40배 정도 많이 걸리지만(최악의 경우 약 8초), 저장 공간 면에서는 약 30배 정도 효율적인 것으로 나타났다. 또한, XML 뷰를 유지하는 것이 그렇지 않은 경우보다 질의 초기 수행 시간(실험 결과 중 초기 뷰 질의 처리 시간)은 많이 걸렸지만(VI의 경우 약 2~3배, MV의 경우 약 1.3~3.5배), 질의 재수행 시간은 적게 걸렸다(VI의 경우 약 0.02~0.8배, MV의 경우 약 0.01~0.04배).

본 논문의 향후 연구로는 XML 뷰 정의를 위해 사용하는 XQuery에서 고려되지 않은 특성 및 속성 연산을 지원하기 위한 확장과 뷰 인덱스의 저장 공간 및 뷰 검색 성능의 개선을 위한 최적화 방안에 대한 연구가 필요하다.

참 고 문 헌

[1] S. Abiteboul, "On Views and XML," Proc. ACM Symp. on Principles of Database System, 1999, pp. 1-9.

[2] S. Abiteboul et al., "Active Views for Electronic Commerce," Proc. Int'l Conf. on VLDB, 1999, pp. 138-149.

[3] S. Kim and H. Kang, "XML Query Processing Using Materialized Views," Proc. Int'l Conf. on Internet Computing, Jun. 2001, pp. 111-117.

[4] Y. Papakonstantinou and V. Vianu, "DTD Inference for Views of XML Data," Proc. of 19th ACM SIGACT-SIGMOD-SIGART Symp. on PODS, 2000.

[5] Lucie Xyleme, "A Dynamic Warehouse for XML Data of the Web," IEEE Data Eng. Bulletin, Vol. 24, No. 2, Jun. 2001, pp. 40-47.

[6] http://www.xyleme.com

[7] M. Fernandez et al., "SilkRoute: Trading between Relations and XML," Proc. the 9-th WWW Conf., 2000, pp. 723-746.

[8] J. Shanmugasundaram et al., "Efficiently Publishing Relational Data as XML Documents," Proc. Int'l Conf. on VLDB, 2000, pp. 65-76.

[9] J. Chen et al., "NiagaraCQ: A Scalable Continous Query System for Internet Databases," Proc. ACM SIGMOD Int'l Conf. on Management of Data, 2000, pp. 379-390.

[10] M. Stonebraker, "Implementation of Integrity Constraints and Views by Query Modification," Proc. ACM SIGMOD Int'l Conf. on Management of Data, 1975, pp. 65-78.

[11] A. Gupta and I. Mumick, "Materialized Views: Techniques, Implementations, and Applications," MIT Press, 1999.

[12] N. Roussopoulos, "An Incremental Access Method for ViewCache: Concept, Algorithms, and Cost Analysis," ACM Trans. on Database Systems, Vol. 16, No. 3, Sep. 1991, pp. 535-563.

[13] P. Valduriez, "Join Indices," ACM Trans. on Database Systems, Vol. 12, No. 2, Jun. 1987, pp. 218-246.

[14] S. Abiteboul et al., "Incremental Maintenance for Materialized Views over Semistructured Data," Proc. Int'l Conf. on VLDB, 1998, pp. 38-49.

[15] D. Suciu, "Query Decomposition and View Maintenance for Query Languages for Unstructured Data," Proc. Int'l Conf. on VLDB, 1996, pp. 227-238.

[16] Y. Zhuge and H. Garcia-Molina, "Graph Structured Views and Their Incremental Maintenance," Proc. Int'l Conf. on Data Engineering, 1998, pp. 116-125.

[17] P. Buneman et al, "Programming Constructs for Unstructured Data," Proc. DBPL, 1995.

[18] R. Cattell et al., "The Object Database Standard: ODMG-93," Morgan Kaufmann, 1994.

[19] Y. Papakonstantinou et al., "Object Exchange across Heterogeneous Information Sources," Proc. Int'l Conf. on Data Engineering, 1995, pp. 251-260.

[20] J. McHugh et al., "Lore: A database Management System for Semistructured Data," SIGMOD Record, Vol. 26, No. 3, Sep. 1997, pp. 54-66.

[21] S. Abiteboul et al, "The Lorel Query Language for Semistructured Data," J. of Digital Libraries, Vol. 1, No. 1, Nov. 1996.

[22] R. Goldman and J. Widom, "DataGuides: Enabling Query Formulation and Optimization in Semistructured Databases," Proc. VLDB, 1997, pp. 436-445.

[23] J. McHugh et al., "Indexing Semistructured Data," Tech. Report, Dept. of Computer Science, Stanford Univ., 1998.

[24] T. Milo and D. Suciu, "Index Structures for Path Expressions," Proc. ICDT, 1999.

[25] V. Apparao et al., "Document Object Model Level 1 (Second Edition) (W3C Working Draft)," http://www.w3.org/TR/REC-DOM-Level-1/, 1998.

[26] D. Chamberlin et al., "XQuery 1.0: An XML Query Language," http://www.w3.org/TR/xquery, 2001.

[27] D. Chamberlin et al., "Quilt: an XML Query Language for Heterogeneous Data Sources," In Lecture Notes in Computer Science, Springer-Verlag, Dec. 2000.

[28] World Wide Web Consortium. "XML Path Language (XPath) Version 2.0," W3C Working Draft,

December, 2001. See http://www.w3.org/TR/xpath20/
[29] J. Robie et al., "XML Query Language(XQL)," http://www.w3.org/TandS/QL/QL98/pp/xql.html, 1998.
[30] A. Deutsch et al., "XML-QL: A Query Language for XML," http://www.w3.org/TR/1998/NOTE-xml-ql/, 1998.
[31] SourceForge, "Kweelt," http://kweelt.sourceforge.net/, 2000.

김 영 성

1997년 중앙대학교 컴퓨터공학과 졸업 (공학사). 1999년 중앙대학교 컴퓨터공학과 석사과정 졸업 (공학석사). 1999년~현재 중앙대학교 컴퓨터공학과 박사과정 재학중. 2001년 1월~2002년 11월 (주) 케이포엠 재직. 2002년 12월~현재 한국컴퓨터통신(주) 재직중. 관심분야는 XML, 웹 데이타베이스

강 현 철

1983년 서울대학교 컴퓨터공학과 졸업 (공학사). 1985년 U. of Maryland at College Park, Computer Science(M.S.). 1987년 U. of Maryland at College Park, Computer Science(Ph.D.). 1988년~현재 중앙대학교 컴퓨터공학과 교수. 관심분야는 이동 데이타베이스, 웹 데이타베이스, DBMS 저장 시스템