Journal of Korea Multimedia Society Vol. 7, No. 1, January 2004(pp. 1-11)

관계 데이터베이스를 이용한 XML 문서 저장시스템 설계

신병주†, 진 민††, 이종학†††

요 약

본 논문에서는 관계 데이터베이스를 이용한 XML 문서 저장시스템을 제안한다. XML 구조와 관계 데이터베이스 스키마의 구조적 불일치로 인하여 XML 문서의 저장을 위해서는 별도의 처리과정이 필요하다. 본 연구에서는 DTD가 있는 XML 문서의 관계 데이터베이스 저장을 다루고 있다. 기존의 *Shared Inlining* 방법과 *Hybrid Inlining* 방법을 보완한 릴레이션의 단편과 과도한 조인 연산을 줄여주는 *Association Inlining* 방법을 제안하고 실제 실험을 통하여 성능이 개선되었음을 보인다. DTD를 단순화시켜 이로부터 저장구조 정보를 추출하고 관계 데이터베이스의 저장구조로 사상시키기 위하여 기존의 맵 클래스를 확장하였다. XML 문서의 분석을 통하여 다중값 엘리먼트, 여러 개의 상위 엘리먼트를 갖는 엘리먼트, 순환 구조를 갖는 엘리먼트 등 다양한 형태의 구조에 대해 맵 클래스를 정의하였으며 추출 저장구조 정보와 맵 클래스를 이용하여 XML 형태의 맵 파일을 생성한다.

Design of a Storage System for XML Documents using Relational Databases

Byung-Joo Shin†, Min Jin††, Jong-Hak Lee†††

ABSTRACT

In this paper, we propose a storage system for XML documents using relational databases. Additional processing is required to store XML documents in the relational databases due to the discrepancy between XML structures and relational schema. This study aims to store XML documents with DTD in the relational databases. We propose the *association inlining* that exploits *shared inlining* and *hybrid inlining* and avoids relation fragments and excessive joins. Experiments show some improvements in the performance with the proposed method. The information of the storage structures is extracted from the simplified DTD. Existing map classes are extended in order to map various structures of XML to relational schema. Map classes are defined for various structures such as elements with multiple values, elements with multiple super elements, and elements with recursive structures through analyzing XML documents. Map files that are XML structures and used in generating SQL statements are created by using the extracted information of storage structures and map classes.

Key words: XML, RDBMS, Association Inlining, Map Files(맵 파일)

※ 교신저자(Corresponding Author) : 신병주, 주소 : 경상남도 마산시 월영동 449(631-701), 전화 : 055)249-2653, FAX : 055)248-2554, E-mail : challenger@hawk.kyungnam.ac.kr
접수일 : 2002년 10월 28일, 완료일 : 2003년 6월 9일
† 준회원, 경남대학교 대학원 컴퓨터공학과 박사과정
†† 종신회원, 경남대학교 정보통신공학부 교수
(E-mail : mjin@kyungnam.ac.kr)
††† 종신회원, 대구 가톨릭대학교 컴퓨터정보통신공학부 교수
(E-mail : jhlee11@cu.ac.kr)
※ 이 연구는 경남대학교 2000년도 학술연구조성비 지원에 의하여 수행되었음.

1. 서 론

XML은 정보 통신망의 발달과 함께 데이터 표현 및 교환의 표준으로 중요한 위치를 차지하게 되었다 [8,12]. 이에 따라 전세계적으로 XML을 지원하기 위한 다양한 연구들이 진행중이고, 많은 XML 지원시스템들이 개발되었다. 그 중에서도 XML 데이터를 효율적으로 데이터베이스에 저장하기 위한 많은 연구가 진행되고 있다. XML 저장시스템은 파일시스

템, RDBMS, OODBMS, XML 전용 저장시스템 등으로 구분할 수 있다. 이 중에, RDBMS는 기존의 데이터 뿐만 아니라, XML 데이터와 같은 반구조적 데이터가 공존할 수 있고 대용량 데이터베이스에 대해 복잡한 질의의 처리 속도가 타 저장시스템에 비해 우수하다. 또한, 널리 사용되고 있다는 장점이 있다[1,2,5].

그러나, XML은 엘리먼트들의 계층 구조로 이루어져 있으나, 관계 데이터베이스는 릴레이션의 평면 구조로 구성되어 있다. 따라서, 이러한 XML과 관계 데이터베이스간의 구조적 불일치 문제는 XML 데이터를 RDBMS에 저장함에 있어 별도의 처리과정을 요구한다. 그리고, XML 문서를 관계 데이터베이스에 저장하는 방법에 따라 질의 처리의 효율성 저하, 데이터의 중복, 데이터의 단편화 문제 등이 발생한다[3,8].

따라서, 본 논문은 XML 데이터를 관계 데이터베이스에 저장함에 있어 구조적 차이로 인해 발생하는 문제점을 해결하기 위한 효율적인 저장 시스템을 제안한다. 이 저장 시스템은 DTD가 있는 XML 문서를 대상으로 하며, DTD로부터 관계 스키마 생성을 위한 정보를 추출한다.

본 논문의 구성은 다음과 같다. 2장에서는 XML 문서를 관계 데이터베이스에 저장하기 위한 기존의 연구에 대해 살펴보고, 3장에서는 관계 스키마 생성 방법을 제안하고 그 제안 방법에 의한 XML 문서 저장 시스템을 설계한다. 4장에서는 본 논문에서 제안한 관계 스키마 생성 방법인 *Association Inlining* 방법과 기존의 방법을 비교하고 마지막으로 5장에서는 결론를 맺는다.

2. 관련연구

XML 문서를 관계 데이터베이스에 저장하기 위한 다양한 방법들은 모델 사상 방법과 구조 사상 방법으로 분류할 수 있다[3,10].

모델 사상 방법은 XML 문서의 구조 정보에 관계없이 관계 스키마를 정의하여 XML 문서를 그 스키마에 맞게 저장하는 방법으로 XML 문서의 구조 정보가 저장 구조에 영향을 주지 않는 방법이다. 모델 사상 방법에서는 XML 문서를 레이블이 있는 그래프로 표현하여 각 엘리먼트는 노드로, 하위 엘리먼트와의 관계는 에지로 표현한다. 이 때, 노드는 엘리먼트의 구별을 위해 식별자로 표현되고 에지의 레이블은 하위 엘리먼트의 이름으로 표현된다. 그리고, XML 문서의 값들은 단말노드로 표현한다. 그리하여 그래프의 에지가 저장되는 방법에 따라 에지(*Edge*) 방법, 바이너리(*Binary*) 방법, 유니버셜(*Universal*) 방법으로 구분할 수 있고, 값을 저장하는 방법에는 데이터 형에 따라 별도의 테이블에 저장하거나 에지를 저장하는 테이블의 컬럼에 저장하는 방법이 있다[2].

에지 방법은 모든 에지들을 하나의 테이블에 저장한다. 이 방법은 XML 문서가 대용량인 경우 에지의 수가 많아지므로 질의 처리를 위해 많은 자체조인 연산이 필요하다.

바이너리 방법은 같은 레이블을 가지는 에지들만을 그룹화시켜 하나의 테이블에 저장하는 방법이다. 즉, XML 문서에서 같은 이름을 가지는 엘리먼트나 애트리뷰트들을 한 테이블에 저장하는 방법이다. 바이너리 방법의 단점은 XML 문서상에서 이름이 다른 엘리먼트가 많을 경우 테이블의 수가 많아지게 되고 여러 테이블에 분산되어 저장됨에 따라 질의 처리시 많은 조인 연산이 필요하게 된다는 것이다.

하나의 테이블에 모든 에지들을 저장하는 유니버셜 방법은 바이너리 테이블들을 외부 조인한 것과 동일한 결과의 테이블이 생성된다. 이 방법은 널 값을 갖는 속성의 수가 많아지게 되고, 정규화가 이루어지지 않음으로써 데이터 중복으로 인한 저장공간의 낭비가 심하다.

값을 저장하는 방법 중에 데이터형에 따라 에지들과 분리하여 저장하는 방법은 질의 처리시 항상 조인 연산이 발생하는 문제점이 있다. 값을 에지들이 저장되는 테이블에 함께 저장하는 방법은 모든 데이터형을 나타낼 수 있는 속성을 생성하므로, 널 값을 갖는 속성의 수가 많아진다.

모델 사상 방법과 달리, 구조 사상 방법은 DTD 의존적인 저장 방법이다. XML 문서의 구조를 표현하는 DTD 분석을 통해 XML 문서 저장을 위한 구조 정보를 추출하여 관계 스키마에 사상하는 방법이다. 구조 사상 방법에서 XML 문서를 관계 데이터베이스로 사상하는 기본적인 방법은 XML 문서의 엘리먼트는 관계 스키마의 릴레이션으로, 애트리뷰트는 릴레이션의 속성으로 사상하는 것이다. 그러나, XML 문서의 엘리먼트, 애트리뷰트의 구조와 관계 스키마의

릴레이션, 속성의 구조는 항상 일치하는 것은 아니다. XML 문서의 구성요소를 관계 스키마로 사상하는 방법으로 *Basic Inlining* 방법, *Shared Inlining* 방법, *Hybrid Inlining* 방법 등이 제안되었다[8].

Basic Inlining 방법은 모든 엘리먼트에 대해 각각의 릴레이션으로 사상하고, 하위 엘리먼트는 사상된 릴레이션의 속성으로 사상한다. 그리고, 다중값을 갖는 엘리먼트와 순환구조를 갖는 엘리먼트의 저장을 위해 별도의 릴레이션을 생성한다. 이 방법은 과도한 릴레이션이 생성되고 하나의 엘리먼트가 여러 릴레이션에 나누어져 저장됨으로써 간단한 질의라 하더라도 여러 릴레이션에 대해 질의를 처리한 후 결과값들을 합병해야 하는 어려움이 발생한다.

Shared Inlining 방법에서는 각 엘리먼트는 하나의 릴레이션에서만 표현되고 릴레이션은 다른 릴레이션이 공유할 수 있다. DTD 그래프 상에서 *in-degree* 값이 0이거나 2 이상일 경우에는 릴레이션으로 사상하고, 1일 경우에는 릴레이션의 속성으로 사상한다. 그리고 다중값을 갖는 엘리먼트와 순환구조를 갖는 엘리먼트는 *Basic Inlining* 방법과 마찬가지로 별도의 릴레이션을 생성하여 저장한다. 이 방법은 일부 엘리먼트들을 속성으로 사상함으로써 *Basic Inlining* 방법에서 발생하는 과도한 릴레이션 생성 문제를 해결하고 한 엘리먼트가 여러 릴레이션에 중복되어 표현됨에 따라 발생하는 질의 수 증가 문제를 해결하였지만, 릴레이션의 공유로 인해 질의 처리시 조인 연산의 횟수가 증가하는 단점이 있다.

Hybrid Inlining 방법은 *Shared Inlining* 방법에서 공유되는 릴레이션을 속성으로 사상하여 조인 연산 횟수가 증가되지 않도록 한다. 즉, 다중값을 갖는 엘리먼트, 순환구조를 갖는 엘리먼트가 아니면서 *in- degree* 값이 2이상인 엘리먼트들은 속성으로 사상하여 *Shared Inlining* 방법의 단점인 조인 연산의 횟수 증가 문제를 해결한다. 그러나, *Hybrid Inlining* 방법은 엘리먼트가 여러 릴레이션의 속성으로 사상함으로써 특정 질의 처리시 여러 릴레이션에서 처리된 값들은 합쳐야하는 복잡한 과정이 필요하게 된다.

3. XML 문서 저장시스템 설계

3.1 시스템 개요

그림 1은 본 논문에서 설계한 관계 데이터베이스를 이용한 XML 문서 저장시스템의 구성도이다. 복잡한 DTD를 단순화시키는 DTD 변환기, DTD 그래프를 생성하는 DTD 파서, 관계 스키마 생성 정보를 맵 클래스와 결합하여 맵 팡일을 생성하는 맵 파일 생성기, 관계 스키마 생성을 위한 SQL 문을 생성하는 SQL 문 생성기 등의 모듈로 이루어진다.

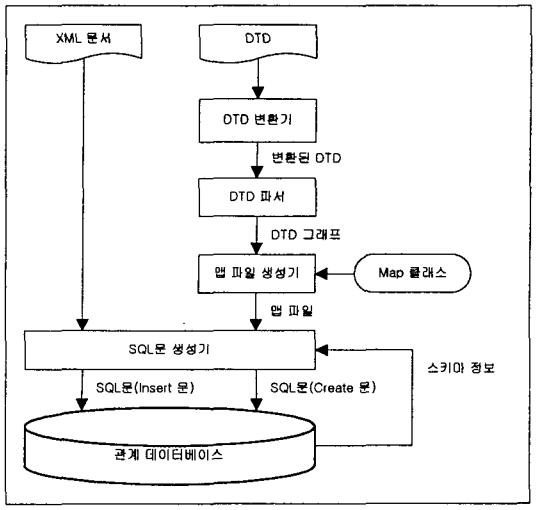

그림 1. XML 문서 저장시스템의 구성도

3.2 DTD 변환

3.2.1 DTD 단순화

일반적으로 XML 문서는 복잡한 구조의 DTD를 갖는다. 복잡한 구조의 DTD로부터 관계 스키마를 생성하는 것은 어려움이 발생하므로 원래의 DTD 구조 정보의 손실없이 DTD를 따르는 XML 문서의 저장과 질의가 가능하도록 DTD를 단순화한다. 단순화 방법들은 다음과 같다[3,8].

1) ( )에 의해 내포된 정의는 ( )를 제거한다. 이 때, "|" 연결자는 "," 연결자로 대체한다.

(PHONE | EMAIL) --> PHONE*, EMAIL**

2) 복수 개의 발생 지시자는 단일 발생 지시자로 통합한다.

(PHONE+) --> PHONE* (NAME?)* --> NAME**

(EMAIL+)? --> EMAIL (EMAIL?)? --> EMAIL?*

3) 같은 이름을 가진 복수 개의 하위 엘리먼트는 하나로 그룹화시킨다. 이 때, 발생 지시자의 통합 규칙은 (2)의 통합 규칙과는 다르다.

NAME, PHONE, NAME? --> NAME*, PHONE*

NAME?, PHONE, NAME? --> NAME, PHONE*
NAME, PHONE, NAME --> NAME+, PHONE

3.2.2 DTD 파싱

본 논문에서는 DTD로부터 저장 방법을 추론하여 관계 스키마를 생성하고 XML 문서를 저장하게 된다. 따라서, 먼저 DTD 파싱을 통해서 DTD의 구조를 표현하는 DTD 그래프를 생성한다. 그림 3은 그림 2의 DTD의 구조를 표현한 DTD 그래프이다. DTD 그래프에서는 엘리먼트, 애트리뷰트, 반복 지시자 등을 노드로 표현하고 각 엘리먼트와 애트리뷰트 간의 관계는 에지로 나타낸다. 그리고, 노드 표현시 DTD 상에서 같은 이름을 가진 엘리먼트는 한 노드로 표현되고, 애트리뷰트는 이름과 관계 없이 DTD에 나타나는대로 DTD 그래프에 표현된다[8].

DTD 그래프는 연결 리스트로 표현하는데, DTD 그래프의 각 노드에 대해 한 개의 연결 리스트가 존재한다. 연결 리스트는 각 노드의 정보를 갖고 있는

```
<!ELEMENT book (booktitle, author)>
<!ELEMENT booktitle (#PCDATA)>
<!ELEMENT article (title, author*, contactauthor)>
<!ELEMENT contactauthor EMPTY>
<!ATTLIST contactauthor authorID IDREF IMPLIED>
<!ELEMENT monograph (title, author, editor)>
<!ELEMENT editor (monograph*)>
<!ATTLIST editor name CDATA #REQUIRED>
<!ELEMENT title (#PCDATA)>
<!ELEMENT author (name, address)>
<!ATTLIST authorid ID #REQUIRED>
<!ELEMENT name (#PCDATA)>
<!ELEMENT firstname (#PCDATA)>
<!ELEMENT lastname (#PCDATA)>
<!ELEMENT address ANY>
```

그림 2. DTD 예

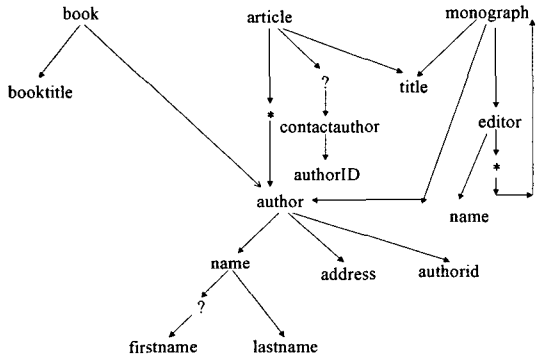

그림 3. DTD 그래프

필드들을 갖는다. 리스트 *i*에 있는 노드들은 노드 *i*의 자식 노드들을 나타낸다. 각 리스트들은 헤드 노드를 가짐으로써, 임의의 노드에 대한 신속한 접근이 가능하도록 순차적으로 번호를 부여한다. 그림 4는 DTD 그래프를 표현하는 연결 리스트를 위한 선언문이다.

그림 5는 그림 1의 DTD를 연결 리스트로 표현한 것이다. *nodeid*의 값이 0인 리스트의 자식 노드에는 XML 문서상에서 루트 노드를 표현하여 매핑 정보를 추출하기 위한 순회를 시작할 노드를 나타낸다. 그리고 자식 노드들을 나타내는 연결 리스트는 *childnode* 구조체를 사용한다. *childnode* 구조체는 자식 노드의 식별자 값을 갖는 *nodeid* 필드와 다른 자식 노드와 연결을 위한 *link* 필드로 구성된다.

```
# define MAX_VERTICES 15 /* DTD 그래프의 노드 총개수*/
typedef struct childnode *node_pointer;
typedef struct childnode { /* 임의의 노드에 대한 자식 노드 */
    int nodeid;
    node_pointer link;
};
typedef struct {    /* 임의의 노드에 대한 정보 표현 */
    int nodeid;    /* 노드 식별자 */
    char nodename[];    /* 노드명 */
    char keyword;    /* 엘리먼트와 애트리뷰트의 구별 */
    char type[];    /* 엘리먼트와 애트리뷰트의 종류 */
    char operator;    /* "*", "+", "?" 등의 반복 지시자 표현 */
    char default[];    /* 애트리뷰트의 디폴트 값 표현 */
    int indegree;    /* DTD 그래프 상의 in-degree 값 */
    boolean visited;
    childnode link;  /* 자식 노드를 표현하는 리스트의 연결 */
} node;
node graph[MAX_VERTICES + 1];    /* 헤드 노드 */
```

그림 4. DTD 그래프 표현을 위한 연결 리스트 구조

3.3 관계 스키마 생성

3.3.1 Association Inlining 방법

XML 문서의 엘리먼트와 속성을 관계 스키마로 표현하기 위하여 *Association Inlining* 방법을 제안한다. 이 방법은 기존의 *Shared Inlining* 방법과 *Hybrid Inlining* 방법의 장점을 수용할 수 있도록 한다.

Association Inlining 방법에서 XML 문서의 엘리먼트를 관계 스키마의 릴레이션이나 속성으로 사상하는 기준은 DTD 그래프에서 노드의 *in-degree* 값이다. 그림 3의 DTD 그래프[8]에서 *book* 엘리먼트처럼 *in-degree* 값이 0일 경우에는 상위 엘리먼트가 존재하지 않으므로 릴레이션으로 사상한다. 그리고 *in-degree* 값이 1인 경우에는 상위 엘리먼트에 포함

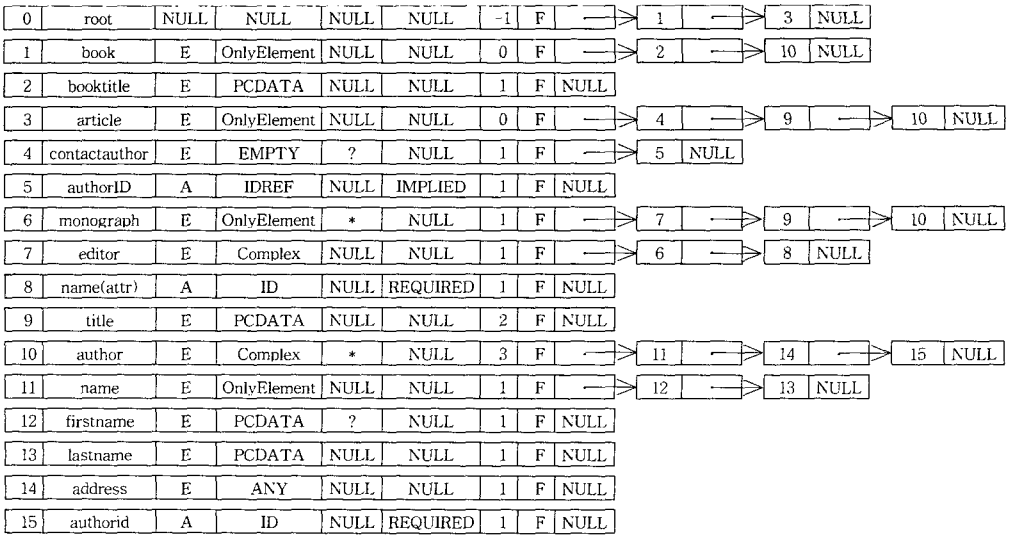

그림 5. DTD 그래프를 표현한 연결 리스트

시켜 관계 스키마의 속성으로 사상한다.

in-degree 값이 2 이상일 경우에는 별도의 릴레이션을 생성하는 것을 원칙으로 하지만 예외적인 경우가 있다. *in-degree* 값이 2 이상일지라도 *title* 엘리먼트처럼 단말 노드라면 새로운 릴레이션을 생성하는 것은 의미가 없다. 새로운 릴레이션을 생성한 경우 오히려 조인 연산의 증가만을 초래한다. 따라서, 이 경우에는 새로운 릴레이션을 생성하는 것보다 상위 엘리먼트에 속성으로 표현한다. 또한, *author* 엘리먼트와 *title* 엘리먼트를 비교해 보자. 두 엘리먼트는 모두 *in-degree* 값이 2 이상이다. 그런데, *author* 엘리먼트는 릴레이션으로, *title* 엘리먼트는 릴레이션의 속성으로 사상한다. 이와 같이, 두 엘리먼트의 사상 방법이 다른 이유는 상위 엘리먼트와의 상관관계가 다르기 때문이다. 즉, *author* 엘리먼트는 상위 엘리먼트의 속성으로 포함시키는 것보다 독립된 객체로 보는 것이 타당하다. 즉, 독립적인 객체의 성격이 있고, 추가적으로 하위 엘리먼트나 속성을 필요로 한다. 그러나, *title*의 경우는 독립된 객체로 보기 어렵다. 텍스트로 이 엘리먼트의 성질을 기술할 수 있기 때문이다. 따라서, *in-degree* 값이 2 이상일 경우에는 상위 엘리먼트와 하위 엘리먼트의 상관관계의 분석을 통해 릴레이션 또는 속성으로 표현한다. 다중값을 갖는 엘리먼트와 순환 구조를 갖는 엘리먼트는 *in-degree*의 값과는 무관하게 별도의 릴레이션을 생성하여 표현한다.

복수 개의 상위 엘리먼트를 가지거나 다중값을 갖는 엘리먼트, 순환구조를 갖는 엘리먼트 등을 릴레이션으로 사상할 경우 다른 릴레이션과 관계를 갖는다. 다른 릴레이션과의 관계 표현을 위해 *parentID*와 *parentCode* 속성을 추가한다. 엘리먼트가 릴레이션의 속성으로 사상되었을 경우 이 엘리먼트가 XML 문서의 루트가 되는 경우의 질의 처리를 위해 *isroot* 라는 속성을 추가한다[8]. 그림 6은 그림 3의 DTD 그래프를 통하여 생성된 릴레이션이다.

Book (id,BookTitle.isroot,BookTitle,doc_id)
Article (id,ContactAuthor.isroot,Authorid,Title.isroot,Title, doc_id)
Monograph (id,ParentID,ParentCode,Title.isrrot,Title Editor.isroot,Name,doc_id)
Author (id,ParentID,ParentCode,Name.isroot,FirstName, FirstName.isroot,LastName,LastName.isroot, Address.isroot,Address,Authorid,doc_id)

그림 6. DTD 그래프에 의한 관계 스키마의 생성 예

3.3.2 구조정보 저장을 위한 관계 스키마

XML 문서가 여러 릴레이션에 분리되어 저장됨에 따라 XML 문서가 가지는 구조 정보가 손실된다. 따라서, XML 문서의 구조 정보를 저장하여 원래의 XML 문서의 복원이 가능하게 하고 질의 처리시 데이터베이스에 대한 접근 횟수를 감소시키기 위하여 DTD와 각 릴레이션에 대한 정보를 저장하는 *ElementInfo* 릴레이션과 *AttributeInfo* 릴레이션을

추가적으로 생성한다.

3.4 맵 파일 생성

맵 파일 생성기는 DTD 그래프로부터 구조정보를 추출하여 관계 스키마로의 사상 방법을 결정한 후 해당 맵 클래스를 호출하여 XML로 표현된 맵 파일을 생성한다[10,11]. 맵 클래스는 본 논문에서 사용한 관계 스키마 생성 알고리즘에 따라 DTD 그래프 상의 각 노드들을 관계 스키마로 사상하는 방법을 정의한 클래스이다. 그림 7은 맵 파일 생성기에서 DTD 그래프를 표현한 연결 리스트를 분석하여 그 결과값을 바탕으로 맵 클래스를 호출하는 알고리즘을 보여주고 있다. 호출된 맵 클래스에 의해 생성된 각 인스턴스들은 최종적으로 조합되어 XML 문서 형태의 하나의 맵 파일을 생성한다.

DTD는 XML에 대한 규칙을 정의하는데 있어 XML과 다른 문법을 사용한다. 따라서 일관성이 없고, 데이터베이스에 사상하기가 어려우며, 사용자가 원하는 형태를 정의해서 사용할 수가 없어 유연성이 떨어진다. 따라서, 맵 파일 생성기에서는 분석된

```
char parent[]; char BaseElement[];
main() { TraverseNode(); }
TraverseNode() {
   for( int nodeid = 0; nodeid <= MAX_VERTICS; nodeid++ )
      if( graph[i].visited == False ) NodeAnalyzer( nodeid );
}
NodeAnalyzer( int nodeid ) {
   switch( graph[nodeid].indegree ) {
      case -1 :
         TraverseChildNode( nodeid );
         break;
      case 0 :
         parent = graph[nodeid].name;
         MapToRelation( nodeid, 0 );
         break;
      case 1 :
         if( graph[nodeid].keyword == "E" ) {
            BaseElement = graph[nodeid].name;
            if( IsRecursiveElement( nodeid ) ) {
               parent = graph[nodeid].name;
               MapToRelation( nodeid, 2 );
            }
            else if( IsSetValued( nodeid ) ) {
               parent = graph[nodeid].name;
               MapToRelation( nodeid, 1 );
            }
            else if( graph[nodeid].type == 'OnlyElement' )
               MapToAttribute( nodeid, 0 );
            else
               MapToAttribute( nodeid, 1 );
         }
         else
            MapToAttribute( nodeid, 2 );
            break;
      default :
         if( IsSetValued( nodeid ) ) {
            parent = graph[nodeid].name;
            MapToRelation( nodeid, 2 );
         }
         else if( graph[nodeid].link == NULL )
            MapToAttribute( nodeid, 1 );
         else {
            Analyze the relationship of the element-subelement;
            If( the relationship is independent ) {
               parent = graph[nodeid].name;
               MapToRelation( nodeid, 2 );
            }
            else
               MapToAttribute( nodeid, 1 );
         }
         break;
   }
}
TraverseChildNode( int nodeid ) {
   for( ptr = graph[nodeid].link; ptr; ptr = ptr -> link )
      NodeAnalyzer( ptr -> nodeid );
}
```

```
MapToRelation( int nodeid, int maptype ) {
   switch( maptype ) {
      case 0 :
Call the map class for creating a relation corresponding to the parameter
maptype;
         break;
           :
   }
   graph[nodeid].visited = True;
   TraverseChildNode( nodeid );
}
MapToAttribute( int nodeid, int maptype ) {
   switch( maptype ) {
      case 0 :
Call the map class for creating only an isroot attribute;
         break;
      case 1 :
Call the map class for creating an isroot attribute and an attribute;
         break;
      case 2 :
         switch( graph[nodeid].type ) {
            /* Call the map class corresponding to the type of attribute */
            case 'CDATA' :
                  :
               break;
                  :
         }
         break;
   }
   graph[nodeid].visited = True;
}
Boolean IsRecursiveElement( int nodeid ) {
   IsRecursion = False;
   for( ptr = graph[nodeid].link; ptr; ptr = ptr -> link ) {
      if( BaseElement == graph[ptr -> nodeid].name &&
                                    graph[ptr -> nodeid].keyword == "E"
) {
         IsRecursion = True;
         break;
      }
      else
         IsRecursion = IsRecursiveElement( ptr -> nodeid );
   }
   return IsRecursion;
}
Boolean IsSetValued( int nodeid ) {
   if( graph[nodeid].operator == "*" || graph[nodeid].operator == "+" )
      return True;
   else
      retrun False;
}
```

그림 7. 맵 클래스를 호출하는 알고리즘

ElementInfo (ID,Elename,Operator,Attribute,
ElementType,Parent,RelationFlag,doc_id)
Attribute (ID,ElementName,AttributeName,
AttributeType,doc_id)

그림 8. ElementInfo 릴레이션과 AttributeInfo 릴레이션

DTD 정보를 바탕으로 XML 문법을 사용한 맵 파일을 생성한다. 그림 9는 그림 1의 DTD를 통해 생성된 맵 파일의 예이다. 이를 위해 호출되는 맵 클래스를 살펴본다.

• 맵 파일의 루트 엘리먼트와 릴레이션 사상을 위한 맵 클래스

```
<!ELEMENT XMLToRDBMS
                    (IgnoreRoot*, RelationMap+)>
<!ELEMENT IgnoreRoot (#PCDATA)>
<!ELEMENT RelationMap
          ((ToTable | ToSpecialTable), AttributeMap+)>
<!ATTLIST RelationMap
                    ElementName CDATA #REQUIRED>
```

• 데이터베이스에 생성될 테이블과 컬럼 정의 맵 클래스

```
<!ELEMENT ToTable (PrimaryKey, ParentID?)>
<!ATTLIST ToTable TableName CDATA #REQUIRED>
<!ELEMENT PrimaryKey EMPTY>
<!ATTLIST PrimaryKey (PCDATA)>
          Generate (Yes | No) #REQUIRED
          isRoot (Yes | No) #REQUIRED>
<!ELEMENT ParentID (ParentCode?, ParentElement+)>
<!ATTLIST ParentID ColumnName CDATA #REQUIRED>
<!ELEMENT ParentCode (PCDATA)>
<!ELEMENT ParentElement (PCDATA)>
<!ATTLIST ParentElement
                    ForeignKey CDATA #REQUIRED>
```

• *Mixed*와 *Any Element*의 저장을 위한 맵 클래스

```
<!ELEMENT ToSpecialTable (SubElement, RelationMap+)>
<!ATTLIST ToSpecialTable
          TableName CDATA #REQUIRED
          PrimaryKey CDATA #REQUIRED>
<!ELEMENT SubElement
        (ForeignKey, TableName, LookupKey, Sequence)>
<!ATTLIST SubElement TableName CDATA #REQUIRED>
<!ELEMENT ForeignKey (PCDATA)>
<!ELEMENT TableName (PCDATA)>
<!ELEMENT LookupKey (PCDATA)>
<!ELEMENT Sequence (PCDATA)>
```

• 애트리뷰트의 사상을 정의한 맵 클래스

```
<XMLToRDBMS>
 <RelationMap ElementName="book">
  <ToTable TableName="book">
   <PrimaryKey Generate="Yes" isRoot="No">bookid</PrimaryKey>
  </ToTable>
  <AttributeMap>
   <ToColumn AttributeName="booktitle">
    <ColumnName isRoot="Yes">booktitle</ColumnName>
   </ToColumn>
  </AttributeMap>
 </RelationMap>
 <RelationMap ElementName="article">
  <ToTable TableName="article">
   <PrimaryKey
          Generate="Yes" isRoot="No">articleid</PrimaryKey>
  </ToTable>
  <AttributeMap>
   <IgnoreElement ElementName="contactauthor" isRoot="Yes">
  </AttributeMap>
  <AttributeMap>
   <ToColumn AttributeName="authorID">
    <ColumnName isRoot="No">authorID</ColumnName>
    <AttributeRef RefTable="author">authorid</AttributeRef>
   </ToColum>
  </AttributeMap>
  <AttributeMap>
   <ToColumn AttributeName="title">
    <ColumnName isRoot="Yes">title</ColumnName>
   </ToColumn>
  </AttributeMap>
 </RelationMap>
 <RelationMap ElementName="author">
  <ToTable Tablename="author">
   <PrimaryKey Generate="No" isRoot="No">authorid</PrimaryKey>
   <ParentID ColumnName="ParentID">
    <ParentCode>ParentCode</ParentCode>
    <ParentElement ForeignKey="bookid">book</ParentElement>
    <ParentElement ForeignKey="articleid">article</ParentElement>
    <ParentElement
          ForeignKey="monographid">monograph</ParentElement>
   </ParentID>
  </ToTable>
  <AttributeMap>
   <IgnoreElement ElementName="name" isRoot="Yes">
  </AttributeMap>
  <AttributeMap>
   <ToColumn AttributeName="firstname">
    <ColumnName isRoot="Yes">firstname</ColumnName>
   </ToColumn>
  </AttributeMap>
  <AttributeMap>
   <ToColumn AttributeName="lastname">
    <ColumnName isRoot="Yes">lastname</ColumnName>
   </ToColumn>
  </AttributeMap>
  <AttributeMap>
   <ToColumn AttributeName="authorid">
    <ColumnName isRoot="No">authorid</ColumnName>
   </ToColumn>
  </AttributeMap>
 </RelationMap>
 <RelationMap ElementName="monograph">
  <ToTable Tablename="monograph">
   <PrimaryKey
          Generate="Yes" isRoot="Yes">monographid</PrimaryKey>
   <ParentID ColumnName="ParentID">
    <ParentCode>ParentCode</ParentCode>
    <ParentElement
          ForeignKey="monographid">monograph</ParentElement>
   </ParentID>
  </ToTable>
  <AttributeMap>
   <IgnoreElement ElementName="editor" isRoot="Yes">
  </AttributeMap>
  <AttributeMap>
   <ToColumn AttributeName="name">
    <ColumnName isRoot="Yes">name</ColumnName>
   </ToColumn>
  </AttributeMap>
  <AttributeMap>
   <ToColumn AttributeName="title">
    <ColumnName isRoot="Yes">title</ColumnName>
   </ToColumn>
  </AttributeMap>
 </RelationMap>
</XMLToRDBMS>
```

그림 9. 그림 1의 DTD에 의한 맵 파일 생성 예

```
<!ELEMENT AttributeMap
        (IgnoreElement | ToColumn | ToSeperateTable)>
<!ELEMENT IgnoreElement EMPTY>
<!ATTLIST IgnoreElement
            ElementName CDATA #REQUIRED
            isRoot (Yes | No) #REQUIRED>
<!ELEMENT ToColumn (ColumnName, AttributeRef*)>
<!ATTLIST ToColumn
                AttributeName CDATA #REQUIRED>
<!ELEMENT ColumnName (PCDATA)>
<!ATTLIST ColumnName isRoot (Yes | No) #REQUIRED>
<!ELEMENT AttributeRef (PCDATA)>
<!ATTLIST AttributeRef
            RefTable CDATA #REQUIRED>
<!ELEMENT ToSeperateTable (RelationMap)>
```

3.5 SQL문 생성기

저장되는 XML 문서와 맵 파일 생성기에 의해 생성된 맵 파일의 DOM 트리를 각각 생성한다. 생성된 DOM 트리는 데이터베이스에 XML 문서를 저장하는 SQL문을 생성할 수 있도록 해 준다. 먼저 맵 파일에 대한 DOM 트리로부터 데이터베이스에 실제로 생성될 테이블과 컬럼을 정의하는 SQL문을 생성한다. 그리고 XML 문서에 대한 DOM 트리를 통해 XML 문서의 데이터베이스 저장을 위한 SQL문을 생성한다. SQL문 생성기에 의해 생성된 두 종류의 SQL 문의 실행을 통해 데이터베이스에 XML 문서의 저장이 이루어지게 된다. 그림 10는 그림 9의 맵 파일을 DOM 트리 분석을 통해 생성되어지는 SQL 문장이다. 각 속성의 데이터 형은 기본적으로 기본키와 외래키의 역할을 하는 속성은 *int*형으로, XML 데이터 저장을 위한 속성들은 *varchar(50)*, *isroot* 속성은 *char(1)*로 설정한다.

4. 관계 스키마 생성 방법의 비교 분석

본 논문에서 제안한 XML 문서 저장을 위한 관계 스키마 생성 방법인 *Association Inlining* 방법은 기존의 *Shared Inlining* 방법과 *Hybrid Inlining* 방법의 장점을 수용하는 방법이다. *Shared Inlining* 방법은 각 엘리먼트를 하나의 릴레이션에서만 표현함으로써 조인 연산의 횟수가 증가하는 문제점이 있고 *Hybrid Inlining* 방법은 조인 연산의 횟수를 줄이기 위해 엘리먼트를 여러 릴레이션의 속성으로 사상함으로써 질의가 증가하는 문제점이 있다. 그림 11, 12, 13은 질의 처리시 발생하는 질의의 수와 조인 연산의 횟수를 *Association Inlining* 방법과 기존의 두 방법을 비교한 결과이다. 관계 스키마 생성 방법의 비교를 위해 10개의 DTD를 이용하였으며 경로 길이가

```
CREATE TABLE Book (
        BookId              int NOT NULL,
        BookTitle           varchar(50) NOT NULL,
        BookTitleisroot     char(1) NOT NULL,
        PRIMARY KEY (BookId) )
CREATE TABLE Article (
        ArticleId           int NOT NULL,
        AuthorId            int NULL,
        ContactAuthorisroot char(1) NOT NULL,
        Title               varchar(50) NULL,
        Titleisroot         char(1) NOT NULL,
        PRIMARY KEY (ArticleId) )
CREATE TABLE Author (
        AuthorId            int NOT NULL,
        ParentId            int NULL,
        ParentCode          varchar(50) NULL,
        Nameisroot          char(1) NOT NULL,
        FirstName           varchar(50) NULL,
        FirstNameisroot     char(1) NOT NULL,
        LastName            varchar(50) NULL,
        LastNameisroot      char(1) NOT NULL,
        PRIMARY KEY (AuthorId) )
CREATE TABLE Monograph (
        MonographId         int NOT NULL,
        ParentId            int NULL,
        ParentCode          int NULL,
        Editorisroot        char(1) NOT NULL,
        Name                varchar(50) NULL,
        Nameisroot          char(1) NOT NULL,
        Title               varchar(50) NULL,
        Titleisroot         char(1) NOT NULL,
        PRIMARY KEY (MonographId) )
ALTER TABLE Article
ADD FOREIGN KEY (AuthorId) REFERENCES Author
ALTER TABLE Author
ADD FOREIGN KEY (ParentId) REFERENCES Monograph
ALTER TABLE Author
ADD FOREIGN KEY (ParentId) REFERENCES Article
ALTER TABLE Author
ADD FOREIGN KEY (ParentId) REFERENCES Book
ALTER TABLE Monograph
ADD FOREIGN KEY (ParentId) REFERENCES Monograph
```

그림 10. 그림 9의 맵 파일에 의해 생성된 SQL 문

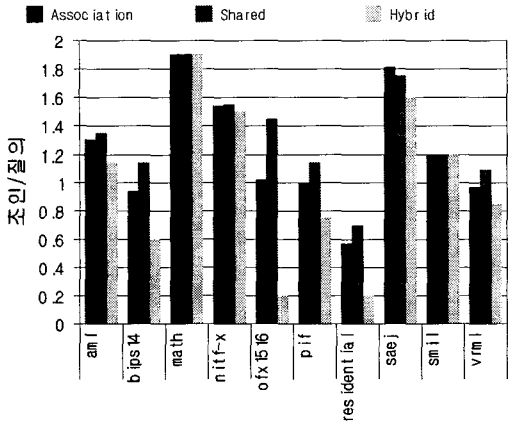

그림 11. 질의당 발생하는 조인 연산 횟수

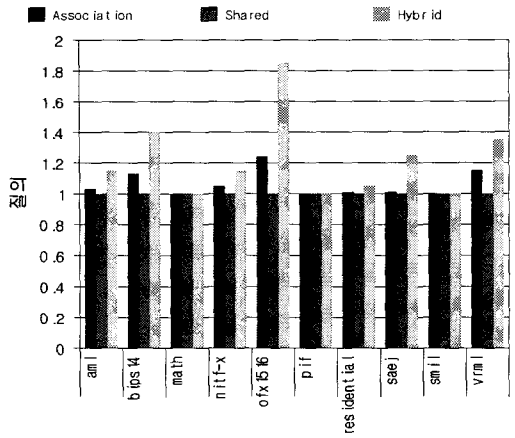

그림 12. 질의문의 갯수 비교

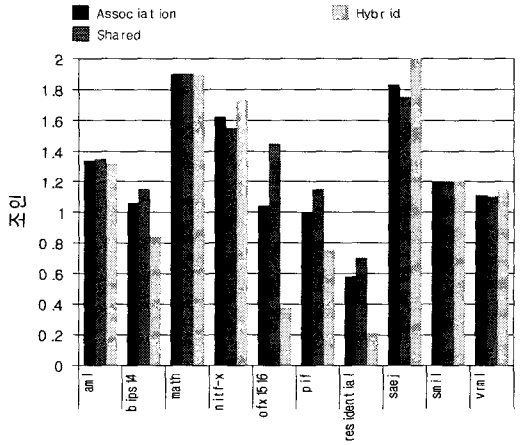

그림 13. 총 조인 연산 횟수

3인 경우에 각 질의 당 발생하는 조인 연산의 횟수, 질의의 수, 전체 조인 연산의 횟수를 각각 비교하였다. 그림 11은 *Association Inlining* 방법이 *Shared Inlining* 방법에 비해 조인 연산의 횟수가 적음을 보여주고 있다. *Shared Inlining* 방법에 비해 약 92.11%의 조인 연산이 발생하는 것을 알 수 있다. 그림 12는 *Association Inlining* 방법이 *Hybrid Inlining* 방법에 비해 질의의 수가 12.95% 감소하는 것을 보여주고 있다. 그림 13은 *Association Inlining* 방법이 질의를 처리하는데 필요한 총 조인 연산의 수가 *Shared Inlining* 방법에 비해 약 95.34%임을 보여주고 있다. 따라서, *Association Inlining* 방법이 *Shared Inlining* 방법에 비해서 조인 연산의 횟수가 감소하고 *Hybrid Inlining* 방법에 비해서는 질의의 수가 감소하는 특징을 갖고 있다고 할 수 있다. 그리고, 이와 같은 특징은 경로 표현식의 경로 길이가 길수록 더욱 뚜렷하게 나타나는 것을 확인할 수 있다. 그러나, 예제의 *math*와 *smil*과 같은 DTD는 관계 스키마 생성 방법과는 무관하게 비슷한 성능 평가 결과를 보여주고 있는데 이는 *Association Inlining* 방법을 비롯한 기존 방법들이 다중값을 갖는 엘리먼트와 순환 구조의 엘리먼트에 대한 처리에 대한 처리 방식이 같기 때문이다.

5. 결 론

XML 문서와 관계 데이터베이스의 구조적 불일치로 인해 XML 문서를 관계 데이터베이스에 저장할 경우 별도의 처리과정이 필요하다. 본 논문에서는 DTD 기반의 XML 문서를 관계 데이터베이스에 저장하는 시스템을 설계하였다. 질의 처리시 과도한 조인이 발생되는 *Shared Inlining* 방법의 단점과 질의의 수가 증가되는 *Hybrid Inlining* 방법의 단점을 개선하는 *Association Inlining* 방법을 제안하였다. DTD는 다양한 데이터 타입을 정의할 수 없어 데이터베이스와의 원활한 연계가 어렵고 사용자가 원하는 형태를 정의하여 사용할 수 없기 때문에 유연성이 떨어지는 단점을 가지고 있다. 또한, DTD는 XML에 대한 규칙을 정의함에 있어 XML과 다른 문법을 사용하기 때문에 일관성을 유지할 수가 없다. 따라서, 본 논문에서는 관계 데이터베이스에 저장하는데 필요한 정보를 DTD 분석을 통해 추출하여 이를 XML 문서 형태의 맵 파일로 생성한다. 맵 파일 생성시에 참조하는 맵 클래스 파일을 사용자가 유연하게 정의할 수 있도록 하였다. 또한, 다중값을 갖는 엘리먼트와 복수 개의 상위 엘리먼트를 갖는 엘리먼트, 순환 구조를 갖는 엘리먼트, *Mixed element* 등에 대해 저장 형태를 정의함으로써 XML 문서가 관계 데이터베이스로 충실하게 사상될 수 있도록 하였다.

실제 실험을 통하여 본 논문에서 제안된 *Association Inlining* 방법의 성능을 기존의 *Shared Inlining*과 *Hybrid Inlining* 방법과 비교하였는데 *Shared Inlining* 방법에 비해 조인의 수가 줄어들었고, *Hybrid Inlining* 방법에 비해서는 질의문의 수가 감소하였다. 향후 XML 데이터의 구조정보에 대한 질의의 효율적인 처리를 위해 구조정보를 저장하는 인덱스 스킴과의 결합 방안에 대한 연구가 필요하다.

또한, DTD 그래프에서 *in-degree* 값이 2 이상인 엘리먼트의 관계 스키마로의 사상 방법을 결정하는 체계적인 연구가 필요하다.

참 고 문 헌

[1] A. Deutsch, M. Fernandez, D. Suciu, "Storing Semi-Structured Data with STORED", SIGMOD Conference, Philadelphia, May 1999.

[2] D. Florescu, D. Kossmann, "Storing and Querying XML Data using an RDBMS", Data Engineering Bulletin, Vol. 22, No. 3, 1999.

[3] D. Lee, W.W. Chu, "Constraints-perserving Transformation from XML Document Type Definition to Relational Schema", Proceedings of the International Conference on Conceptual Modeling, pages 323-338, 2000.

[4] D. Kroenke, Database Processing, Prentice Hall, 2000.

[5] E. Bertino, B. Catania, "Integrating XML and Databases", IEEE, August 2001.

[6] H. Jiang, H. Lu, W. Wang, J. Xu Yu, "Path Materialization Revisited: An Efficient Storage Model for XML Data", The 2nd Australian Institute of Computer Ethics Conference, Canberra, 2000.

[7] I. Tatarinov, S. D. Viglas, K. Beyer, J. Shanmugasundaram, E. Shekita, C. Zhang, Storing and Querying Ordered XML Using a Relational Database System, ACM SIGMOD, Madison, Wisconsin, June 2002.

[8] J. Shanmugasundaram, K. Tufte, G. He, C. Zhang, D. DeWitt, J. Naughton, "Relational Databases for Querying XML Documents: Limitations and Opportunities", VLDB, Edinburg, Scotland, 1999.

[9] K. Williams, M. Brundage, P. Dengler, J. Gabriel, A. Hoskinson, M. Kay, T. Maxwell, M. Ochoa, J. Papa, M. Vanmane, Professional XML Databases, Wrox Press, 2000.

[10] M. Yoshikawa, T. Amagasa, "XRel: A Path-Based Approach to Storage and Retrieval of XML Documents Using Relational Databases", ACM Transactions on Internet Technology, Vol. 1, No. 1, August 2001, pp. 110-141.

[11] R. Bourret, "Mapping DTDs to Databases:, www.xml.com/pub/a/2001/05/09/dtdtodbs.html, 2001.

[12] R. Bourret, C. Bornhovd, A. Buchmann, "A Generic Load/Extract Utility for Data Transfer Between XML Documents and Relational Dtabases", TR-DBS99-1, DVS, Dep.CS, Darmstadt U. of Technology, Germany, Dec. 1999.

[13] T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler, "Extensible Markup Language (XML) 1.0 (Second Edition)", www.w3.org/TR/REC-xml.

[14] 신병주, 진민, 정민수, "XML 문서의 관계 데이터베이스 저장", 한국정보처리학회 추계학술발표논문집 제8권 제2호, pp.55-58, 2001

신 병 주

2001년 경남대학교 전자계산학과 학사
2003년 경남대학교 컴퓨터공학과 석사
2003년~현재 경남대학교 컴퓨터공학과 박사과정

관심분야 : 데이타베이스, 데이타 모델링, XML

진 민

1982년 서울대학교 계산통계학과 학사
1984년 한국과학기술원 전산학과 석사
1997년 University of Connecticut 컴퓨터공학과 박사
1985년~현재 경남대학교 정보통신공학부 교수

관심분야 : 데이타베이스, 객체지향데이타베이스, 웹데이타베이스, XML, 데이타 모델링

이 종 학

1982년 경북대학교 전자공학과 학사
1984년 한국과학기술원 전산학과 석사
1997년 한국과학기술원 전산학과 박사
1998년~현재 대구가톨릭대학교 컴퓨터정보통신공학부 교수

관심분야 : 데이타베이스 시스템, 객체 데이터베이스, 트랜잭션 프로세싱, 지리정보 시스템