효율적인 GML 문서 저장을 위한 저장 스키마의 설계 및 성능평가

왕태웅° 장재우
전북대학교 컴퓨터 공학과
{twwang°, jwchang}@dblab.chonbuk.ac.kr

Design and Performance Analysis of Storage Schema for Efficient Storing GML Documents

Tae-Woong Wang° Jae-Woo Chang
Dept. of Computer Engineering, Chonbuk National University

요 약

최근 LBS(Location Based Service) 및 텔레매틱스 응용의 효과적인 지원을 위해, 도로, 철도와 같은 공간 데이터베이스에 관한 연구가 활발히 수행중에 있다. 공간 네트워크 데이터베이스 연구는 지리 정보의 교환 표준으로 제시된 GML(Geographic Markup Language)을 지원하는 연구가 필수적이다. 기존 XML 저장스키마는 공간지리정보 표현에 적합하지 않기 때문에, 본 논문에서는 공간 지리정보를 포함한 GML을 저장하기 위한 새로운 저장스키마를 제안한다. 아울러 제안하는 저장 스키마의 효율성을 평가하기 위해 디스크 기반 및 메모리 기반의 하부 저장 시스템을 통해 성능평가를 수행한다.

1. 서 론

최근 LBS나 텔레매틱스 기술의 핵심은 공통적으로 지리정보를 이용한 서비스를 수행한다. OGC(Open GIS Consortium)는 네트워크, 응용 혹은 플랫폼의 형식에 관계없이 지리정보의 교환 표준으로 GML(Geographic Markup Language)을 채택하였다[1]. GML은 피쳐(Feature)라고 불리는 지리적인 실체를 통해서 지리정보를 표현하고 있으며, 공간적인 피쳐와 비공간적인 피쳐를 포함한 지리정보를 전달하거나 저장하기 위해서 XML형태로 표현한다[2][3]. 일반적인 공간 데이터베이스에서 GML 지원을 위한 연구는 크게 3가지 연구로 요약된다. GML 문서의 파싱(parsing), GML 문서의 저장, GML 질의어 등이다.

본 논문에서는 GML에 관한 3가지 주제 가운데 GML 문서의 저장에 관한 연구를 다룬다. 먼저 기존 XML 저장스키마는 공간지리정보 표현에 적합하지 않기 때문에 본 논문에서는 공간 지리정보를 포함하는 GML을 저장하기 위한 새로운 저장스키마를 제안한다. 아울러 제안하는 저장 스키마의 효율성을 보이기 위해서 디스크 기반 및 메모리 기반의 하부 저장 시스템을 통해 성능평가를 수행한다.

본 논문의 구성은 다음과 같다. 2장에서는 관련연구로 기존 XML과 GML의 저장스키마인 XParent와 GParent를 소개하고 3장에서는 GML 문서를 저장하기 위한 새로운 저장 스키마를 제시한다. 4장에서는 본 논문에서 제안한 저장 스키마를 하부 저장 시스템에 구현하여 성능평가를 실시한다. 마지막으로, 5장에서는 결론 및 향후연구를 제시한다.

2. 관련연구

XML은 차세대 웹 문서의 표준으로 정착되어 다양한 분야의 문서들이 XML을 기반으로 생성되고 있다. 특히 지리정보의 교환 표준으로 XML을 이용하는 연구가 진행되고 있다. 이러한 연구의 결과 OGC(OpenGIS Consortium)에서는 XML을 기반으로 지리 정보의 저장 및 전송을 위한 인코딩 표준으로 GML[1]을 제안하였다. 공간정보와 위치정보를 포함하는 기술들의 상호 운용성(Interoperability)에 목적을 두고 만들어진 OGC 컨소시움은 개방형 지리정보시스템 환경을 위해 지리 정보 데이터와 응용 프로그램 간 표준 인터페이스를 제시하는 것을 목표로 2003년 초에 GML 3.0을 제안하였다.

기존의 GML 저장 스키마는 XParent의 기법을 확장한 것과, XParent를 발전시킨 GParent가 있다[4]. 일반적으로 관계형 데이터베이스에 XML/GML 문서를 저장하기 위한 연구는 모델 사상(Model Mapping) 및 구조 사상(Structure Mapping)기법으로 분류할 수 있다[2]. 모델 사상기법은 정해진 데이터베이스 스키마를 이용하여 DTD나 스키마 문서 없이 XML/GML문서를 저장하는 기법이다. 이 기법은 XML문서를 데이터 모델로 변환하는 과정에서 XML의 구조적 특징과 내용을 문서 독립적으로 모델링하여 저장하지 않는다. 따라서 XML문서 구조의 갱신이 발생할 경우, 이에 따른 데이터베이스 구조정보가 수정되지 않아도 되는 기법이다. 대표적인시스템으로 간선(Edge)기반의 Monet[5]과 정점(Node) 기반의 XParent[6][7], XRel[8] 등이 있다. 관계형 데이터베이스 기술을 기반으로 XML문서를 저장하는 또 다른 기법은

XML문서의 구조 정보를 표현하는 DTD(Document Type Definition)또는 XML 스키마를 기반으로 관계형 데이터베이스의 스키마를 정의하고, 정의된 테이블에 XML문서를 저장하는 XML스키마에 종속적인 구조 사상(Structure Mapping)기법이 있다. 구조 사상 기법은 XML스키마를 기반으로 생성된 관계형 데이터베이스(Relational database)의 스키마(Schema)에 따라 저장이 이루어지기 때문에, 스키마들을 만족하는 XML문서들이 많이 존재하거나 스키마 생성에 사용된 스키마의 수정이 빈번하게 발생하지 않는 경우에 적합하다. 대표적인 연구로는 Shared Inlining/Hybrid Inlining 기법[9]과 비용 기반 접근 방식인 LegoDB[10]가 있다.

모델 사상기법의 여러 가지 방법 중 XParent 방법이 가장 발전된 형태이며 제일 우수하다고 알려져 있다[7]. XParent의 스키마는 다음 (표 1)과 같이 5개의 테이블로 이루어져 있다.

표 1. XParent의 5개 테이블의 내용

테이블 이 름	내 용
LabelPath 테이블	XML 문서의 모든 Path 정보가 루트로부터 모두 포함이 되어 있으며, 각각의 Path 하나에 고유한 PathID를 가지며 루트로부터 단계를 Length로 표현하며 루트 자신은 Length 1을 가진다.
Element 테이블	XML 문서에서 모든 Element와 Path 와의 관계를 나타내는 테이블로 각각의 Element는 고유한 DataID를 가지며, 어떤 Path 부합하는지 가리키기 위해 PathID가 있으며, 같은 Path 일 경우 순위를 나타내는 Order로 구성이 되어있다.
DataPath 테이블	부모-자식 엘리먼트와의 관계를 나타내는 테이블로 각각의 엘리먼트에 자식 엘리먼트가 존재할 때 Parent Data ID 와 Child Data ID로 구성이 되어 있다.
Data 테이블	각각의 엘리먼트에 Data 정보를 담고 있는 테이블로 PathID, DataID 와 실제 Data를 담고 있는 Value로 구성이 되어 있다.
Ancestor 테이블	상위 엘리먼트와의 관계를 나타내는 테이블로 각각의 엘리먼트의 모든 조상들을 Level 별로 표현하고 있다.

XParent를 확장하여 GML문서를 저장하기 위한 저장 스키마로서 GParent의 3가지 방법이 제시되었다.[4] XParent는 엘리먼트에 포함된 데이터를 얻어오기 위해 Element 테이블과 Data2 테이블로 부터 동일한 PathID와 DataID를 갖는 레코드를 검색하기 위해 조인 연산이 불가피하다. 따라서 GParent #1은 이러한 점을 해결하기 위해 XParent의 Data2 테이블과 Element 테이블을 결합하여 하나의 테이블(Data-Element)로 표현하는 방법이다. GParent #1은 LabelPath(PathID, Length, Path), Data-Element(PathID, Ordianl, DataID, Type, Value), DataPath(Parent Data ID, Child Data ID), Ancestor (DataID, Ancestor Data ID, Level), Non-Spatial Index (Value, Type, DataID), Spatial Index(Value, Type, DataID)의 구조로 되어 있다.

한편, XParent는 엘리먼트에 바로 하위 엘리먼트가 있을 경우 이 부분을 처리하기 위해 DataPath 테이블과 Element 테이블과의 조인 연산을 수행해야 한다. GParent #2는 이러한 점을 해결하기 위해 Element 테이블과 DataPath 테이블을 결합하여 하나의 테이블(Element-DataPath)로 표현하는 방법이다. GParent #2는 LabelPath(PathID, Length, Path), Data2(PathID, DataID, Type, Value), Element-DataPath(PathID, Ordinal, DataID, Child Data ID), Ancestor(DataID, Ancestor Data ID, Level), Non-Spatial Index(Value, Type, DataID), Spatial Index(Value, Type, DataID)의 구조로 되어 있다.

마지막으로 XParent는 하위 엘리먼트가 존재해도 Data 테이블의 검색이 필요할 경우 이 부분을 처리하기 위해 Data 테이블 Element 테이블 DataPath 테이블과의 조인 연산이 필요하다. GParent #3은 이러한 점을 해결하기 위해 Data 테이블, Element 테이블과 DataPath 테이블을 결합하여 하나의 테이블(Data-Element-DataPath)로 표현하는 방법이다. GParent #3은 LabelPath (PathID, Length, Path), Data-Element-DataPath(PathID, Ordianl, DataID, Type, Value, Child Data ID), Ancestor(DataID, Ancestor Data ID, Level), Non-Spatial Index(Value, Type, DataID), Spatial Index(Value, Type, DataID)의 구조로 되어 있다.

3. GML 문서 저장을 위한 저장 스키마 설계

본 절에서는 GML 문서를 위한 새로운 저장 스키마를 설계하기 위해, 기존 GML 저장 스키마인 GParent와 XParent의 문제점들을 살펴본다. 첫째, XParent는 GML을 저장하기 위한 공간 지리 정보들의 표현의 한계성과 자료 검색의 한계성을 지니고 있다. 즉 XML을 저장할 수 있는 XParent에는 GML을 저장하기 위한 공간 지리 정보의 형식을 지원하는 Type이 없다. 따라서 XParent 저장스키마는 XML의 모든 데이터의 표현을 Data 테이블에 표현하지만, GML 문서에서 공간 지리정보를 포함한 문서를 저장하기 위해서는, 공간 지리정보에 대한 표현 및 색인도 제공해야만 한다. 둘째, XParent의 Data, Element 테이블은 공통적으로 동일한 PathID 와 DataID를 사용하고, Data, Element, DataPath 테이블은 공통적으로 동일한 DataID를 사용한다. 따라서 Table 간의 중복된 데이터가 많아 저장 공간이 비효율적이다. 셋째, XParent는 테이블 수가 많고 DataID로 검색 시 Data, Element, DataPath 3종류의 테이블과 많은 조인 연산을 필요로 하여 검색 성능이 저하된다. 마지막으로, GML 문서를 위한 기존의 저장 스키마는 여러 개의 GML문서를 저장할 경우 문서를 구분할 수 있는 ID가 없다. 따라서 검색 시 각각의 문서에 포함되는 DataID를 구분할 수 있는 방법이 없다. 기존 저장 스키마의 장·단점을 요약하면 (표 2)와 같다.

표 2. 기존 저장스키마의 장·단점 비교

저장스키마	장 점	단 점
XParent	테이블을 의미적으로 구분하여 쉽게 알아볼 수 있음	지리정보들의 표현의 한계성이 있으며 데이터의 중복이 많음 검색 시 여러 개의 테이블을 참조
GParent#1	데이터의 중복과 검색 시 별도의 Data 테이블의 검색을 줄임	DataPath 테이블을 검색해야 완전한 엘리먼트를 얻을 수 있음
GParent#2	데이터의 중복과 검색 시 별도의 DataPath 테이블의 검색을 줄임	Data 테이블을 검색해야 완전한 엘리먼트를 얻을 수 있음
GParent#3	데이터의 중복과 검색 시 별도의 테이블들의 검색을 줄임	하나의 레코드 크기가 증가하므로 삽입/검색 시 성능 저하

위와 같은 문제점을 해결하기 위해 본 논문에서는 XParent를 개선하여 공간 지리정보를 잘 표현하며, 저장 / 검색 형태에 알맞은 새로운 저장스키마를 제안한다. 위에서 제시한 첫 번째 문제를 해결하기 위해 공간 데이터를 저장할 수 있도록 Data Table에 대한 확장이 필요하다. 따라서 Data Table에 'Type'이라는 항목을 추가하고 비 공간(Non-Spatial) 데이터와 공간(Spatial) 데이터의 Data Type을 구분하여 공간 데이터와 비 공간 데이터에 대한 색인을 별도로 제공한다. 아울러 공간 데이터의 검색을 위해 Spatial Table을 생성하여 공간 데이터를 저장하고 Spatial Table의 Value를 키 값으로 공간인덱스를 생성한다. 한편, 위에서 제시한 두 번째 문제를 해결하기 위한 방법으로 데이터 중복을 줄이기 위해 Data, Element, DataPath 3종류의 테이블을 결합하는 방법이 있으나, DataID에 ChildID와 Type이 없는 경우와 DataID에 Data값이 없을 경우 NULL값이 발생하게 되는 문제점(그림 1)이 존재한다. (그림 1)의 예를 보면 DataID &d1은 엘리먼트이며 &d2라는 하위 엘리먼트를 가지고 있을 경우 Type과 Value의 값이 NULL이 된다. 또는 DataID &d5와 같이 Polygon데이터 형으로 DataID의 Type과 Value는 있으나 &d5의 하위 엘리먼트가 없을 경우 ChildID에서 NULL 값을 갖게 된다.

PathID	Ordinal	DataID	ChildID	Type	Value
1	1	&d1	&d2	NULL	NULL
2	1	&d2	&d3	NULL	NULL
3	1	&d3	&d4,&d5	NULL	NULL
4	1	&d4	NULL	ATR	#12
5	1	&d5	NULL	Polygon	229205.83,222644.76 236024.83,228215.87

그림 1. GParent #3의 Data-Element-DataPath 테이블

따라서 위와 같은 두 번째 문제를 해결하기 위해 Element Table과 DataPath Table을 결합하여 Element-DataPath Table(그림 2)을 만드는 방법으로 NULL값을 최소화한다.

PathID	DataID	Ordinal	Type	ChildID
1	&d1	1	DIR	&d2
2	&d2	1	DIR	&d3
3	&d3	1	DIR	&d4, &d5
4	&d4	1	ATR	NULL
5	&d5	1	Polygon	NULL

그림 2. Our Method의 Element-DataPath 테이블

위에서 제시한 세 번째 문제를 해결하기 위해서는 엘리먼트 간의 DataID 검색 시 Element Table과 DataPath Table의 Join 연산은 필수적이다. 따라서 두 Table을 결합하고 Element-DataPath Table을 생성하여 Table간의 Join 연산을 줄이는 방법을 사용한다. 마지막으로 위에서 제시한 네 번째 문제인 DataID의 중복과 GML문서를 구분할 수 없는 문제를 해결하기 위해서 문서ID를 추가하는 방법이 있으나, 이 또한 문서ID 데이터의 중복이 많아지는 문제점이 발생한다. 따라서 본 논문에서는 기존의 DataID형식에 GML문서 번호(DocNum)를 추가하여 DataID의 중복성과 문서ID를 구분할 수 없는 문제점을 해결하는 새로운 DataID 형식을 제안한다. 새로운 DataID는 검색 시 원하는 GML문서의 DataID로 검색하고 싶을 경우 DataID를 Bit & Shift 연산을 수행하여 원하는 문서번호와 DataID를 얻을 수 있는 구조이다. 다음 (그림 3)은 본 논문에서 제안한 새로운 DataID 형식이다.

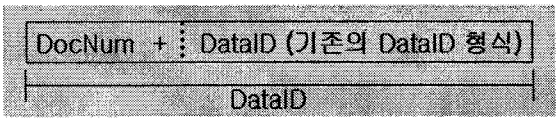

그림 3. DataID의 새로운 형식

위와 같은 문제점들을 해결하여 본 논문에서 설계한 GML 저장 스키마의 구조는 (그림 4)과 같다. (그림 4)에서 음영 처리된 부분은 테이블의 키 값이다. 이러한 4가지 테이블로 구성된 GML 저장 스키마를 Our Method라고 명명한다.

Element-DataPath Table

PathID	DataID	Ordinal	Type	ChildID

LabelPath Table

PathID	Level	Path

Data Table

PathID	DataID	Type	Value

SpatialData Table(Rtree)

DataID	Type	Value

그림 4. 제안하는 GML 저장 스키마

4. 성능평가

본 절에서는 본 논문에서 제시한 Our Method의 효율성을 입증하기 위해 하부저장 시스템 GigaBaseDB[11]및 Gist[12]를 기반으로 구현하여 평가한다. 성능평가 환경은 CPU Intel P4 2.4Ghz, RAM 1GB를 사용하는 Windows Server 2003에서 Visual C++ 7.1로 구현하였다.

효율성 평가에 대한 항목으로 GML 문서의 삽입 시간, 저장 공간 사용량, 엘리먼트 당 검색시간, 공간 데이터 검색 시간을 측정하고 기존 스키마와 비교하였다. 입력 데이터는 세 종류의 데이터를 사용하였다. 문서 단계가 깊고 공간정보가 밀집되어있는 서울 데이터(5개구 포함)와 전라북도의 공간 정보가 담겨있는 전북 데이터 마지막으로 기존 스키마와 성능 비교를 위해 전주시 데이터를 사용하였다. 이 세 가지 GML문서는 (주)Thinkware [13]에서 제공하였다.

4.1 삽입 성능

Our Method의 효율성을 알아보기 위해 디스크 기반의 GigaBase와 메모리 기반인 Gist에 각각 데이터를 삽입한다. (그림 5)는 GML문서들의 삽입시간 성능을 측정한 결과를 보여준다.

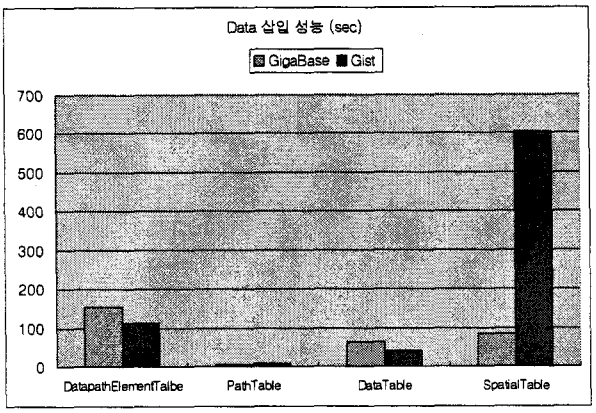

그림 5. GML Data의 삽입 시간

서울 데이터의 Non-Spatial(DataPathElement Table, Path Table, Data Table) 데이터 삽입시간은 GigaBase 221.452sec, Gist 160.858sec 이다. Gist의 삽입 성능이 좋은 이유는 데이터 접근 속도가 빠른 메모리 기반의 하부 저장 시스템이기 때문이다. 한편 서울 데이터의 Spatial(Spatial Table) 데이터 삽입 시간은 GigaBase 82.843sec, Gist 603.406sec 이다. Gist의 삽입 성능이 낮은 이유는 내부에서 지원하지 않는 공간 인덱스를 구성하기 위해 외부의 라이브러리를 사용하는 방법으로 공간 인덱스를 구성하기 때문이다. 전라북도 데이터 역시 서울 데이터의 삽입 시간과 유사한 성능을 보여준다. (전라북도 데이터의 삽입 성능 결과는 서울 데이터와 유사하므로 그림 5에서는 생략한다.)

4.2 저장 공간 사용량

본 절은 Our Method의 저장 공간 사용량 성능평가를 수행한다. (그림 6)는 Our Method를 사용하여 삽입된 GML 문서들의 저장 공간 사용량을 측정한 결과를 보여준다.

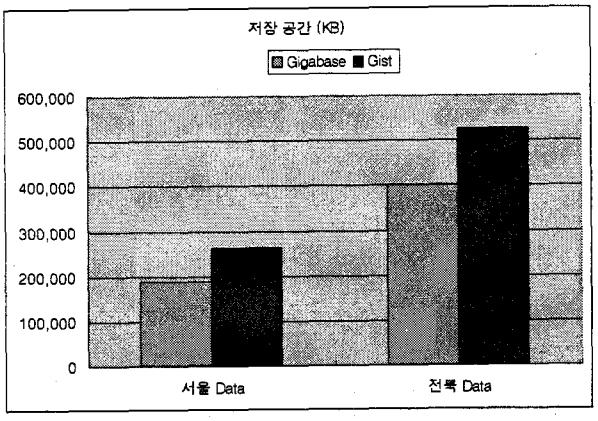

그림 6. 저장 공간 사용량

서울 데이터의 저장 공간 사용량은 GigaBase 187,520 KB, Gist 262,144KB 이다. GigaBase보다 Gist의 저장 공간 사용량이 많은 이유는 Gist가 메모리 기반 하부 저장 시스템이기 때문에 디스크 기반보다 메모리를 사용하기 위한 부가정보가 더 포함되기 때문이다. 전라북도 데이터 역시 서울 데이터와 유사한 성능을 보여준다.

4.3 Non Spatial Data 평균 검색 시간

(그림 7)은 GML문서들의 DataID를 사용하여 비 공간 데이터(DataPathElement Table, Path Table, Data Table)의 검색 시간을 측정한 결과를 보여준다.

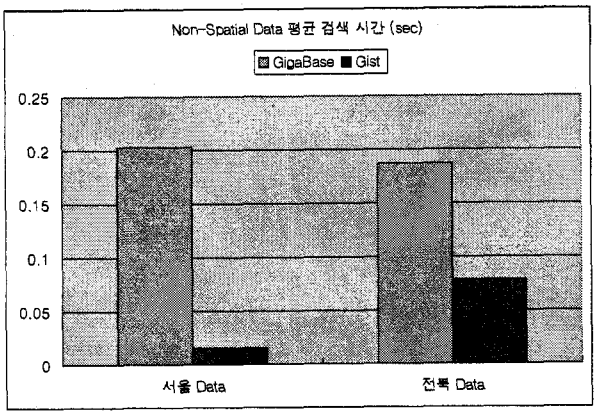

그림 7. Non Spatial Data 평균 검색 시간

검색 방법은 검색하고자 하는 DataID를 입력하면 그 DataID의 하위 Element들까지 검색되어 GML문서의 형식으로 반환된다. 서울 데이터의 Non-Spatial Data 평균 검색 시간은 GigaBase 0.203sec, Gist 0.015sec 이다. 이러한 결과는 디스크 기반의 GigaBase보다 상대적으로 데이터 접근 속도가 빠른 메모리 기반인 Gist의 성능이 좋기 때문이다. 전라북도 데이터 역시 서울 데이터와 유사한 성능을 보여준다.

4.4 Spatial Data 평균 검색 시간

본 절에서는 Our Method의 공간 데이터 검색 성능평가를 수행한다. (그림 8)은 GML문서에 들어있는 공간 데이터들을 삽입 시 구성되는 공간 인덱스에서 공간 질의를

실시하여 검색한 평균 시간을 측정한 결과를 보여준다.

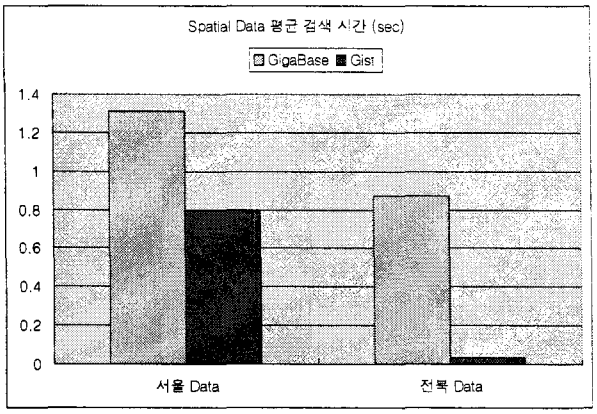

그림 8. Spatial Data 평균 검색 시간

검색 방법은 검색하고자 하는 영역의 Rectangle을 지정하여 영역에 겹쳐있는 공간 데이터들을 반환하게 된다. 서울 데이터의 공간 데이터 평균 검색 시간은 GigaBase 1.312sec, Gist 0.796sec 이다. 이러한 결과는 디스크 기반의 GigaBase보다 상대적으로 데이터 접근 속도가 빠른 메모리 기반인 Gist가 성능이 좋기 때문이다. 전라북도 데이터 역시 서울 데이터와 유사한 성능을 보여준다.

4.5 기존 스키마와 데이터의 삽입 성능 비교

이번 절에서는 Our Method의 효율성을 기존 스키마와 비교하기 위해 디스크 기반의 GigaBase에 구현하여 전주 데이터를 삽입하였다. GParent1은 기존의 스키마에 비해 성능이 낮으므로 이번 성능평가에서 제외하였다. (그림 9)는 GML문서의 삽입시간 성능을 측정한 결과를 보여준다.

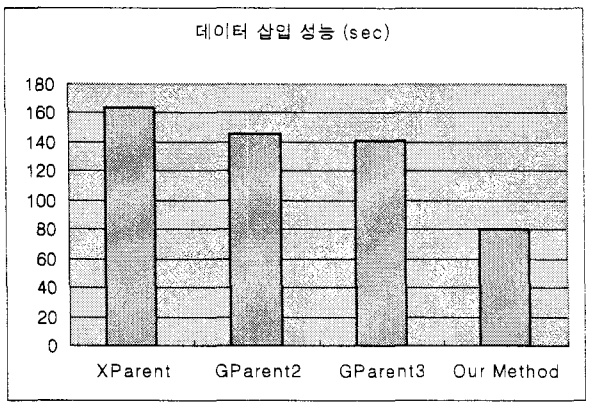

그림 9. 기존 스키마와 데이터 삽입 시간 비교

데이터의 삽입 시간은 XParent 163.701sec, GParent2 145.67sec, GParent 141.592sec, Our Method는 80.378sec 이다. 위와 같은 결과는 기존 스키마에 비해 Our Method의 중복된 데이터가 적기 때문이다. 따라서 Our Method의 삽입 성능이 좋음을 알 수 있다.

4.6 기존 스키마와 저장 공간 사용량 비교

(그림 10)은 기존 스키마와 Our Method를 사용하여 삽입된 GML문서들의 저장 공간 사용량을 측정한 결과를 보여준다.

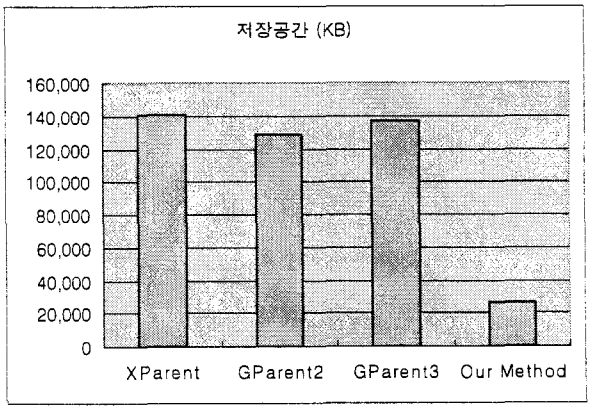

그림 10. 기존 스키마와 저장 공간 비교

전주 데이터의 저장 공간 사용량은 XParent 140,528 KB, GParent2 128,576KB, GParent3 136,688KB, Our Method 26,632KB 이다. Our Method는 기존 스키마에 비해 중복된 데이터와 테이블 수가 적다. 따라서 기존 스키마보다 원본 데이터의 용량이 적은 Our Method의 저장 공간 사용량이 효율적임을 알 수 있다.

4.7 기존 스키마와 Non Spatial Data 검색 시간 비교

(그림 11)에서는 GML문서들의 DataID를 사용하여 기존 스키마와 Our Method의 비 공간 데이터 검색 시간을 측정한 결과를 보여준다.

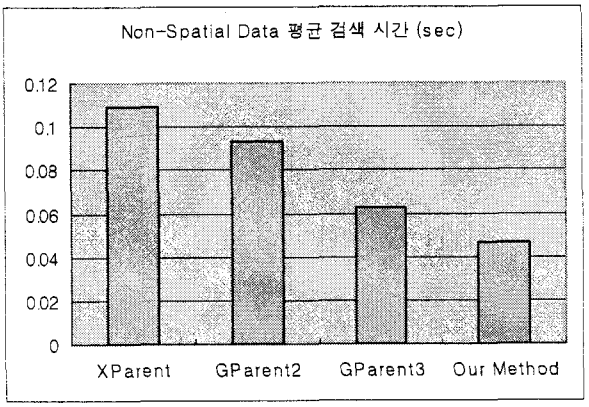

그림 11. 기존 스키마와 Non Spatial Data 검색 시간 비교

검색 방법은 검색하고자 하는 DataID를 입력하면 그 DataID의 하위 Element들까지 검색되어 GML문서의 형식으로 반환된다. 전주 데이터의 검색 시간은 XParent 0.109sec, GParent2 0.093sec, GParent3 0.062sec, Our Method 0.0468sec 이다. 이러한 결과는 DataID의 검색 시 Our Method는 Element Table과 DataPath Table을 결합하여 기존의 스키마보다 Join 연산이 적게 일어나기 때문이다.

4.8 기존 스키마와 Spatial Data 검색 시간 비교

본 절에서는 기존 스키마와 Our Method의 공간 데이터 검색 성능평가를 수행한다. (그림 12)는 GML문서에 들어있는 공간 데이터들을 삽입 시 구성되는 공간 인덱스에서 공간 질의를 실시하여 검색한 평균 시간을 측정한 결과를 보여준다.

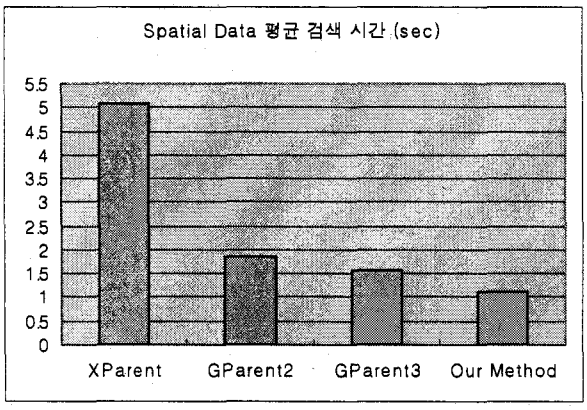

그림 12. 기존 스키마와 Spatial Data 검색 시간 비교

검색 방법은 검색하고자 하는 영역의 Rectangle을 지정하여 영역에 겹쳐있는 공간 데이터들을 반환하게 된다. 전주의 Spatial Data 검색 시간은 XParent 5.093sec, GParent2 1.859sec, GParent3 1.562sec, Our Method 1.109sec이다. Spatial Data 검색은 영역 내에 있는 DataID를 얻기 위해 Join 연산을 해야 한다. 따라서 Join 연산이 적은 구조인 Our Method가 효율적임을 알 수 있다.

5. 결론 및 향후 연구

본 논문에서는 GML 문서의 효율적인 저장과 검색을 위해 XParent를 개선한 GML 저장 스키마인 Our Method를 제시하였다. 또한, 제시한 저장 스키마의 효율성을 평가하기 위해 GigaBaseDB 및 Gist를 기반으로 새로운 저장스키마를 구현하고 평가하였다.

향후 연구로는 본 논문에서 제시한 GML의 저장 스키마를 다양한 상용DBMS에 (Oracle, AltiBase, BerkeleyDB) 구현하여 성능평가를 실시하는 것이다.

Acknowledgment

본 연구는 교육인적자원부, 산업자원부, 노동부의 출연금 및 보조금으로 수행한 최우수실험실 지원사업의 연구결과임.

참고문헌

[1] OGC, "Geography Markup Language(GML) Implementation Specification v3.1.1" , http://www.opengis.net/gml/, 2004.

[2] J. Corcoles et al., "Analysis of Different Approaches for Storing GML Documents", In Proceedings of the 10th ACM International Symposium on Advances in Geographic Information Systems, pp 11-16 2002.

[3] OGC Specifications, "http://www.opengis.org/techno/specs.html", 1999.

[4] 김영국, "GML문서 저장을 위한 저장 스키마 및 하부 저장 시스템의 설계 및 구현", 전북대학교 컴퓨터 공학과 석사 학위 논문, 2006.

[5] A. Schmidt et al., "Efficient Relational Storage and Retrieval of XML Documents" In Proceedings of WEBDB, 2000.

[6] Haifeng Jiang et al., "Path Materialization Revisited: An Efficient Storage Model for XML Data", In Proceedings of the 2nd Australian Institute of Computer Ethics Conference, 2000.

[7] Haifeng Jiang et al. "XParent: An Efficient RDBMS-Based XML Database System.", In Proceeding on the 18th International Conference on Data Engineering, pp 335-336, 2002.

[8] M. Yoshikawa et al., "Xrel: A path-based approach to storage and retrieval of XML Documents using Relational Databases", ACM Transactions on Internet Technology, Vol.1, No.1, pp 110-141, 2001.

[9] Jayavel Shanmugasundaram et al., "Relational databases for querying XML documents: Limitations and opportunities." In Proceedings of 25th International Conference on Very Large Data Bases, pp 302-304, 1999.

[10] P. Bohannon et al., "LegoDB - From XML scheme to relations : a cost-based approach to XML storage", In Proceeding of International Conference on Data Engineering, pp 64-75, 2002.

[11] http://www.garret.ru/%7Eknizhnik/gigabase.html

[12] http://www.fastdb.org/fastdb.html

[13] (주)ThinkWare "http://www.thinkwaresys.com"